ARMY RESEARCH LABORATORY

Simulation and Evaluation of Computation Offloading Algorithms in Battlefield Scenarios

by Brian Swenson, David Doria, and Tamim Sookoor

ARL-TR-6875 March 2014

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Aberdeen Proving Ground, MD 21005-5066

ARL-TR-6875 **March 2014**

Simulation and Evaluation of Computation Offloading Algorithms in Battlefield Scenarios

David Doria and Tamim Sookoor

Computational and Information Sciencies Directorate, ARL

Brian Swenson

School of Electrical and Computer Engineering, Georgia Institute of Technology

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
March 2014	Final	March 2014

4. TITLE AND SUBTITLE
Simulation and Evaluation of Computation Offloading Algorithms in Battlefield Scenarios

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Brian Swenson*
David L. Doria
Tamim I. Sookoor

5d. PROJECT NUMBER
R.0006163.13

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-CIH-S
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-6875

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
*School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332

14. ABSTRACT
Due to their small size and ever-increasing computing capability, mobile devices, such as smart phones, are ideal tools for computation in the battlefield. However, there are some mission-critical applications that cannot be completed in real-time on these devices. Some of these applications can meet their deadlines through computation offloading: sending computationally expensive operations to high-performance computers (HPCs). However, in battlefields it may not be possible to access traditional offloading targets, such as cloud computing services. In this project we examine computation offloading on mobile ad-hoc networks (MANETs) using vehicle-mounted HPCs as offload targets. We use the network simulator ns-3 to model computation offloading in the battlefield. We implemented a suite of extensible ns-3 models, which allow for a wide variety of experiments with MANETs. Using these models, we study a number of aspects of computation offloading including: HPC placement, client offloading strategies, effects of HPC mobility, and variations in network topology. In this report, we describe the overall architecture of the models and provide experimental results on the performance gained by computation offloading when a number of parameters are varied. Our initial results indicate that offloading to mobile HPCs increases the utility of mobile devices as computation platforms in the battlefield.

15. SUBJECT TERMS
computation offload, simulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Brian Swenson
Unclassified	Unclassified	Unclassified	UU	71	19b. TELEPHONE NUMBER (*Include area code*) 410-278-6993

Contents

List of Figures

List of Tables

1. Introduction

The computational capacity of mobile devices, such as smart phones, is continually increasing. Given their small size and computational capability, these devices are becoming ideal tools for increasing the situational awareness of the Warfighter on the battlefield. Yet despite their remarkable characteristics, some applications that would be greatly beneficial to the Warfighter cannot be executed in real-time on these devices. Another significant problem is that these devices have a constrained energy budget and every extra ounce of extra battery weight reduces the amount of armor or ammunition that can be carried by the Warfighter. One possible solution that has been proposed to speed up execution time and extend battery life is computation offloading. Offloading allows computationally expensive operations to be sent from the mobile device to remote high-performance computers (HPCs) where calculations are performed and the results sent back to the mobile device. Possible applications for this technology include: free-form speech recognition, natural language translation, dynamic body language translation from video, and object and facial recognition (*1*).

A frequent assumption in previous studies on computation offloading is that cloud computing services are accessible through stable, high-bandwidth links (*2, 3*). However, on the battlefield access to traditional cloud computing resources such as Amazon EC2(*4*) is usually not available. Even if communication to the cloud is possible, limited communication infrastructure on the battlefield make it unreliable. To avoid these problems, we propose using vehicle-mounted HPCs (*5*) as computation offloading targets. The advantage of using mobile HPCs is that the HPC can move to locations where its effectiveness can be maximized and communication between the mobile devices and the HPC can be performed over an ad-hoc network, which requires no pre-existing communication infrastructure on the battlefield.

The purpose of this project is to create a framework that allows for the evaluation of algorithms for computation offloading to vehicle-mounted HPCs in battlefield scenarios. We use the network simulator ns-3 to model the network communication and create a suite of extensible models, which allow for a wide variety of experiments to be performed. We study a number of aspects of computation offloading including: HPC placement, client offloading strategies, effects of HPC mobility, and network topology.

The rest of this is organized as follows. First, we offer some brief background on ns-3, Transmission Control Protocol (TCP), and Optimized Link State Routing (OLSR). Next, we

detail the models created for this project and how they operate and interact together. Finally, we describe the experiments performed and provide analysis of the results obtained.

2. Background

In this section, we describe the ns-3 simulator, the TCP protocol, and the OLSR protocol.

2.1 ns-3

The ns-3 (*6*) is an open-source, discrete event network simulator for modeling communication networks. The ns-3 provides simulation models for all layers of the networking protocol stack. The simulator is efficient enough that it can easily be run on commodity PCs. We chose to use ns-3 for our simulations because it is one of the most widely used network simulators in academia and it contains the models we need to simulate wireless ad-hoc networks.

2.2 Transmission Control Protocol

For all application level communication in our simulations we used TCP. TCP is a connection oriented, reliable transport protocol. The ns-3 offers models for multiple TCP varients, including TCP Reno, TCP Tahoe, TCP New Reno, and TCP Westwood. The main difference betweeen these varients is the mechanism employed to respond to packet loss during a data transfer. For the experiments in our simulation we used the ns-3 default, TCP New Reno.

TCP is described as a connection oriented protocol because before applications can communicate, they must go though a "handshake" phase where they establish parameters for the ensuing data transfer. During the process they set aside buffers for sending and receiving and initialize TCP state variables.

TCP is referred to as a reliable transport protocol because the byte stream that is received at the destination is guaranteed to be the same as the byte stream that was sent from the sender. TCP ensures that the received stream is without gaps, uncorrupted, and in sequence. It does this through the use of acknowledgement packets and retransmit timers. When a TCP receiver receives a group of packets, it sends an acknowledgment (ack) back to the sender so it knows they have been received. A typical implementation will send back one ack for every two packets that are received. If a TCP sender does not receive an acknowledgement packet within a time period proportional to the measured round trip time between the sender and the receiver, it assumes the

packet was lost and sends it again.

The wireless topologies we modeled in our experiments are ad-hoc wireless networks. The benefit of this type of network is that it does not rely on any pre-existing network infrastructure, such as wireless access points. Wireless networks that do rely on wireless access points are said to be in infrastructure mode. The difference between the two is how data sent from one node to another is routed. See figure 1 for a simple network topology. First assume nodes A, B, and Access Point are in infrastructure mode. Now any data that node A wants to send to node B has to go though the Access Point. For example, if A wishes to send 200 MB of data to B, first the 200 MB of data are transferred to the Access Point and then the 200 MB of data are transferred to B. This is because, in infrastructure mode, the Access Point is responsible for routing all data between the nodes in the network. Therefore, if the Access Point malfunctions or is destroyed, A and B can no longer communicate.

As mentioned previously, the alternative to infrastructure mode is ad-hoc mode. In ad-hoc mode, nodes A and B in figure 2 can communicate with each other directly with no dependency on any external networking infrastructure. Also, if there is another node, node D, which is outside the range of node A but within the range of node B, node A can send data through node B to node D. The network routing tables will update dynamically as the nodes move about. So if node B moves out of range of node A, and node C moves within range of node A, and node C can reach node B, then node A can send data to node B through node C. All of this freedom does not come free however. The routing algorithms that run to dynamically update the route information between nodes can increase overall network latency.

2.3 Optimized Link State Routing protocol

Optimized Link State Routing (OLSR) is a routing protocol used in ad-hoc wireless networks. The protocol is best suited for large and dense ad-hoc networks(*7*). The nodes running the protocol periodically exchange messages in order determine to the network topology and calculate the shortest paths between network nodes. In this way, it is a proactive routing approach as opposed to the on-demand route calculations done by protocols such as Ad-Hoc On-Demand Distance Vector (AODV) routing and Dynamic Source Routing (DSR). One common criticism of the OLSR protocol is that it is possible it may be using computing power and network bandwidth to transfer information about unused links. The obvious benefit is that the routes are available immediately when the sender wishes to send. The result of the OLSR algorithm are optimal routes, in terms of number of wireless hops, between the sender and receiver.

3. Software

In this section, we describe the models that we created and how they interact within the ns-3 simulation. We start by defining terms that are used frequently within this report. We then demonstrate a typical offloading exchange between a client and a server. Finally, we describe specific details about the models.

3.1 Terminology

In ns-3 the most basic modeling unit is the *node*. The simplest way to think of an ns-3 node is as an empty computer shell. To this computer, one can add innards to simulate within the node including network devices, internet protocols, and applications. In the simulations performed for this paper an ns-3 node is modeling either a Warfighter with a mobile computing device or a mobile vehicle with an onboard HPC. An *Application* is a model of a piece of software. Typically, in simulations it is not possible to run real software, like a Web browser. Instead, models are created that capture the apparent properties of the software, including how much data they transmit and how long they run. These applications are installed onto nodes and run during the simulation.

A *client* is an ns-3 node whose purpose in the simulation is to complete *jobs*. To complete a *job* (defined below), the client may break it into multiple chunks, called *tasks*, and offload portions of it to available offload targets, referred to as *servers*. The specific strategy the client uses to complete jobs depends on the type of application that is installed upon the node.

A job is a unit of work performed by the client. Outside of our simulation an example of a job is running a facial recognition algorithm on an image. A job is made up of operations and typically has some set of data that is required to perform the computation. One example of job data is the image in which a facial recognition algorithm attempts to identify faces. When a job is broken into tasks, it may be necessary to also break up the data. For example, if the job is to search an image for a specific pattern, one possibility is to split the image into multiple pieces and send only what is needed to each server. However, if the job is to search a distributed database for an identification based on a photographed image, each server participating in the search would need the entire image.

A *server* is an ns-3 node that is willing to be used as an offload target for clients. The role of a server is to accept and process tasks given to it by the clients in the simulation. In a real scenario,

a mobile vehicle with an HPC would be a server. However, it is possible for one node to function as both a client and a server. For example a mobile phone, albeit typically much slower than an HPC, can be used to complete either a task or an entire job.

3.2 Client Server Communication Protocol

This section describes the communication protocol used between clients and servers. Communication between the client and server is supported by the ComputeHeader class. This header is at the front of all data streams sent between the client and the server and describes the type of data that is being sent/received. Communication between the client and server may be in one of four different stages and the header is used to keep track of what stage they are currently in. See table 1 for a description of each stage. Along with tracking the current stage, specific fields within the header are used at each stage. See table 2 for the fields within the header. The next paragraph provides an example showing how the fields in the ComputeHeader are used.

Figure 3 shows an example of a typical exchange between a client and a single server. Client 1 in figure 3 (a) has a job of 200 operations and a data size of 200 bytes that it wishes to offload. It first sends a ComputeHeader with type StatusRequest to Server 1. In figure 3 (b), Server 1 receives the StatusRequest and replies back to the client with its capacity, which is 2000 operations per second, and its current job queue time, 30 seconds. At this point the client makes its offloading decision. In this example there is only one server and the client wishes to offload, so the decision is trivial. In figure 3 (c), the client sends a task request to the server. In the ComputeHeader it tells the server to perform 200 operations and to expect 200 bytes of data. The client then immediately sends the 200 bytes of data. Since TCP guarantees the data received at the server will be in the order it was sent from the client, the server will receive the ComputeHeader first telling it to expect 200 bytes of data. The server needs to know how much data to expect because for large data transfers the data will not arrive in one packet. Once the server receives the ComputeHeader, it waits until it has received the 200 bytes of data and then places the task into its task queue. Once the task is complete, the server needs to notify the client. In figure 3 (e) the server sends a ComputeHeader to the client with type TaskResponse. When the client receives this TaskResponse ComputeHeader, it knows that the server has finished the task assigned to it.

3.3 Client Application

This section describes the ClientApplication models that we have created. As mentioned previously, the goal for a client is to complete its jobs. The ClientApplication implements the strategy the client uses to complete this task. Each of the different ClientApplication models

Table 1. Types of ComputeHeader.

	Meaning
StatusQuery	A client is making a utilization request from the server.
StatusResponse	A server is responding to a StatusQuery.
TaskRequest	A client is requesting a server complete a task.
TaskResponse	A server is indicating that a requested task is complete.

Table 2. The ComputeHeader Application Header.

Field	**Type**	**Purpose**
Data Size	unsigned 64-bit integer	The amount of data, in bytes, that will be sent
Operations	unsigned 64-bit integer	The number of operations
Capacity	unsigned 64-bit integer	The rate at which the server can perform jobs, in operations per second
Start Time	double	The amount of time until the server finishes all of the jobs on its jobs queue
Type	HeaderType	How the header is being used. See table 1 for a list of possible values.

Table 3. List of ns-3 Attributes in Client.

Attribute	*ns-3 Type*	*Purpose*
Protocol	TypeIdValue	The TypeId of the protocol to use (should use TCP).
NumberOfJobsToRun	UintegerValue	The number of jobs for this client to run.
DataSize	UintegerValue	The size of the data file the client sends to each server it offloads to, in bytes.
JobSize	UintegerValue	The size of the job the client needs to perform, possibly by offloading, in operations.
Stats	Stats	The stats object that the client stores its statisticial information in.

implements a different offloading strategy. The following paragraph gives a brief background on ns-3 application models and then the specifics of the client model are described.

In ns-3, the Application class is a base class, which provides the basic functionality to simulate software running on an ns-3 node. The model of the software to simulate is then provided by the classes, which inherit from this base class. The standard design pattern when developing new application models in ns-3 is to create two classes per new application, a helper class and the new derived application class itself. The helper class provides all of the functionality to configure and install the software model onto the ns-3 node allowing the derived application class to be concerned only with the software that it is modeling.

3.3.1 Setting Up and Configuring a Client Application

ClientApplicationHelpers are used to install and configure an instance of a ClientApplication onto an ns-3 node. Each ClientApplication variant has a corresponding ClientApplicationHelper; see figure 4 for a complete list. Along with installing its corresponding ClientApplication onto a node, the ClientApplicationHelper also is responsible for setting the ns-3 attributes for the ClientApplications it creates. Table 3 shows the attributes the ClientApplication configures.

3.3.2 Running a Client Application

When a client application is started, the first thing that it does is pause for a random period of time. By default this is between 0 and 1. The purpose of doing this is to avoid any unrealistic congestion that would occur with all clients performing initialization, including the TCP 3-way

handshake, in lockstep. While it is important to measure the effects of congestion with multiple clients, in a real scenario it is highly unlikely that every client is going to start at the exact same instant in time.

The next step that each client performs is attempting to connect to the servers that it has knowledge of. When the client application is installed onto an ns-3 node, it is given the Internet Protocol (IP) addresses of all of the servers that it should attempt to connect to when it starts running. In a real-world implementation, a discovery mechanism would need to be present. Since TCP is used exclusively in these models, connecting involves performing the TCP 3-way handshake to create TCP connections to each of the servers. After the connection is made, the client makes a utilization request to each server.

A utilization request is a request to the server to report current state information back to the client node. Currently the response from the server provides two items: compute capability and current delay. These items will be discussed further in section 3.4. Once the utilization requests have been sent, the client waits for a predefined period of time. During this time it is accepting utilization responses back from the servers.

3.3.3 Offloading Strategies

The next step for the client is to make an offloading decision. The offloading strategy for the client is determined by the type of ClientApplication installed on the node. See figure 5 for a list of the available offloading strategies. The following sections describe the ClientApplication variants currently implemented. There is also a brief explanation of how to create new, custom offload strategies.

ClientEqualApplication: The purpose of this application is to spread the job evenly across all servers that respond. This provides a baseline for distributing jobs over multiple offloading targets. The ClientEqualApplication equally partitions a job into equal sized tasks. Each server that responded to a utilization request is sent one of these tasks. Along with the task request, each server will receive a data stream from the client that is the size of the DataSize attribute for the client.

ClientLocalApplication: The purpose of this application is for the client to perform the entire computation locally. This provides a baseline to compare against offloading. For this application to work correctly, the node on which it is installed must also have a server application installed. Also, the InetSocketAddress of the server application running on the client must be added to the client's server address list. Otherwise, no computation will be performed. No data

stream is sent for this type of ClientApplication because it is assumed that the server can access the needed task data locally.

ClientRandomApplication: The purpose of this application is to send the entire job to a randomly selected server. From the servers the client receives utilization responses, which may include the node the ClientApplication is installed on if it also has a ServerApplication installed, the client randomly selects an offload target. It forms its job into one task and sends it to the server along with a data stream that is the size of the DataSize attribute for the client.

ClientOffloadOnlyApplication: The purpose of this application is to ensure that the client offloads the entire job. It sends the entire job as one task to a randomly chosen respondent of a utilization request as long as it is on a different node. This functions similarly to the ClientRandomApplication, however this Application ensures that it will never send to a ServerApplication on the node that the client is running.

ClientArchitectureAwareApplication: The purpose of this application is for the client to perform architecture aware offloading. The client's job is distributed proportionally to the capabilities of the devices that it receives replies from; see figure 6 for a simple example. This type of client application represents an algorithm that only considers hardware profiles of the targets when making offload decisions. Along with the task, like with the other applications, the server also receives a data stream of data. This application can be configured to either send the entire data stream to each server or it can only send each server a percentage of the data. With the percentage option, the server receives the same percentage of the data as the percentage of operations of the overall job. For example, if the server receives 50% of the operations it will receive a data stream that is the size of 50% of the DataSize attribute.

ClientUtilizationAwareApplication: The purpose of this application is for this client to perform architecture and utilization aware offloading. To determine the optimal number of operations to send to each server, this client considers the job division as an integer programming problem. Here we present a brief overview of the approach taken. A full discussion of this algorithm can be found here (*8*). The parameters included in the model are:

- the capability c_i of server i (in $\frac{ops}{sec}$);
- the size of the task j_i assigned to server i (in operations);
- the channel bandwidth W (with units of bytes per second $\frac{B}{sec}$);
- the current task queue Q_i on server i. A server (see section 3.4) keeps a queue of the tasks

assigned to it. This is the amount of time it will take for the server to complete all of the tasks currently on its queue.

- The constant R defines the ratio of the number of bytes that an offload source wants to transmit to the number of operations required (with units of bytes per operation $\frac{B}{ops}$). This ratio R is application specific. For example, in an image-processing application requiring a pixel traversal with a constant computation on each pixel, the size of the computation (in operations) is directly proportional to the size of the image sent (the size of the task, in operations).

Given the above notation, a task's completion time can be written as

$$t_i = j_i \left(\frac{1}{c_i} + \frac{R}{W} \right) + Q_i. \tag{1}$$

Since the tasks are performed in parallel, the job time is going to be equal to the longest task time. Therefore, that is what needs to be minimized. To do this we used LP-solver (*9*) to perform the minimization.

Creating a New Client Application: The ClientApplication model was designed to be flexible and therefore, it is easy to implement a new offloading strategy. The first step is to create the necessary files. The simplest way to do this is to just duplicate one of the other applications, like ClientEqualApplication. There will be four files that need to be duplicated: two for the application (.cc and .h) and two for the application helper (.cc and .h). For example, if duplicating ClientEqualApplication, the four files to duplicate would be: client_equal_application.cc, client_equal_application.h, client_equal_application_helper.cc, and client_equal_application_helper.h. The next step is to rename the files to the new client application name and to change all references within the files to refer to the new class and new helper. Finally, all four of the new files must be added to the CMakeLists.txt file.

The next step is to define the offloading strategy. This is defined in the virtual method DistributeJob in the ClientApplication. Figure 7 gives an example similar to what is implemented in ClientEqualApplication. The information received from the servers is stored in the AssetInformation class; see figure 8 for the Unified Modeling Language (UML) diagram. The nodeId field in this class is a unique key for the server that returned the information. The sockets that connect to the server are indexed in a map by their nodeId; therefore, this key is needed to send the data to the correct location. The capacity and startTime fields were explained

in section 3.4. The hopCount field is the number of hops between the client and the server reported by the OLSR routing protocol. It should be noted that these OLSR tables are being updated throughout the simulation and it is possible for the hopCount to change even though no nodes are moving in the simulation. The value reported in hopCount is the hopCount value in the table when the reply was received from the server.

Lines 5–11 ensure that at least one server has replied to the status request prior to making an offloading decision. In this example, if no replies have been received, the client waits 2 s and the method is called again. Line 14 sets the job start time. This is important to do because this value is needed to correctly calculate the job run time. Line 17 sorts the asset information vector by capacity. For this example it is not necessary, but this can be useful if, for example, we wished to send the entire job to the fastest server. Line 21 sets the number of servers that are being used as offload targets. It is important that this is set correctly because the job will not be considered complete until the client has received the same number of server replies.

Lines 27–46 implement the offloading strategy. In this example the client is splitting the job into N different equal sized tasks, where N is the number of servers that responded. Lines 29–30 create a new ComputeHeader and set the type to Task Request. Line 31 sets the sender ID to the node ID of the client. Line 32 tells the server the number of operations that should be performed for this task. Line 33 tells the server to expect the specified amount of data, in bytes. Lines 34 and 35 create an empty packet and attach the completed Compute Header to it. Finally the header is sent in line 37. Note the use of m_assetInfo[i].nodeId to get the correct socket over which to send.

Now that the server knows the number of operations to perform and the amount of data to expect, it is time to send the data. This is done in lines 42 and 43. Line 42 creates a data packet of the correct size. It is important that the size of the packet created in line 43 is the same as the size specified in line 33. Line 44 sends the data packet to the server. It is important to realize that, just because the data is passed to the socket as one big packet, unless the data size is small, the data will be split into multiple packets as it is transferred to the server at lower levels of the Open Systems Interconnection (OSI) model. This is why it was important to tell the server the number of bytes to expect prior to sending the data in line 33.

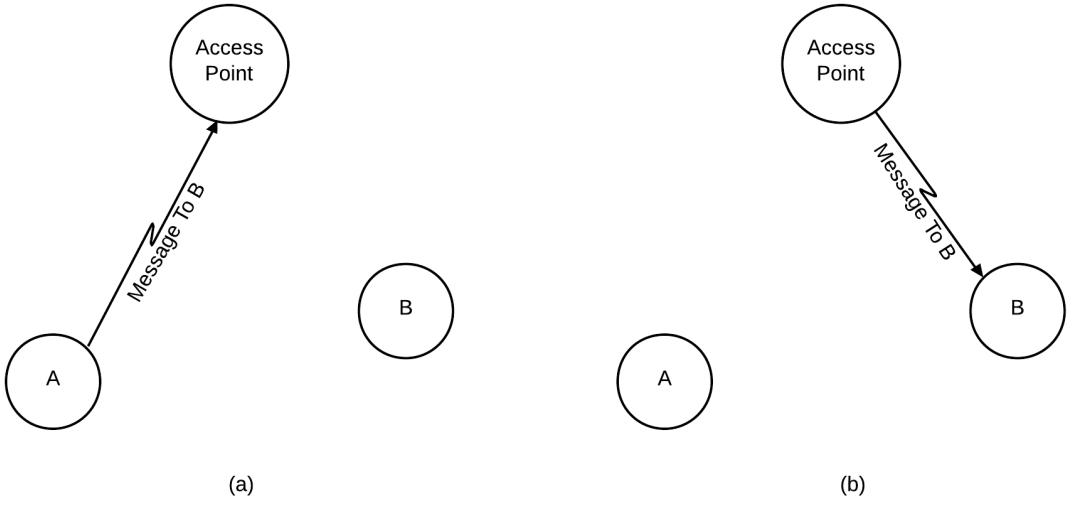

Figure 1. For node A to send a message to nobe B in infrastructure mode, it must go through the Access Point.

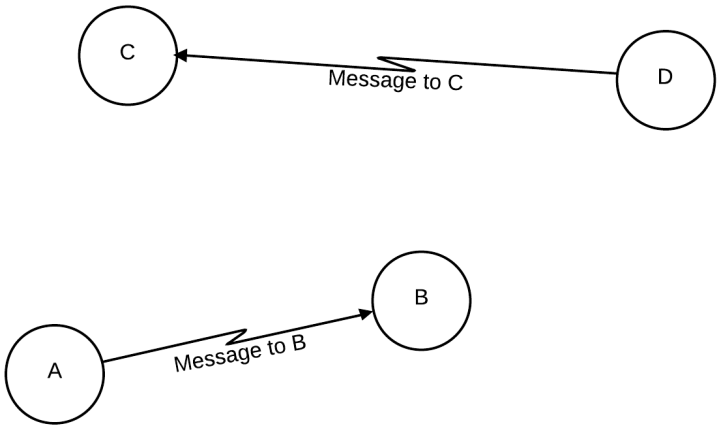

Figure 2. In ad-hoc mode nodes can send messages to each other directly if they are within wireless range.

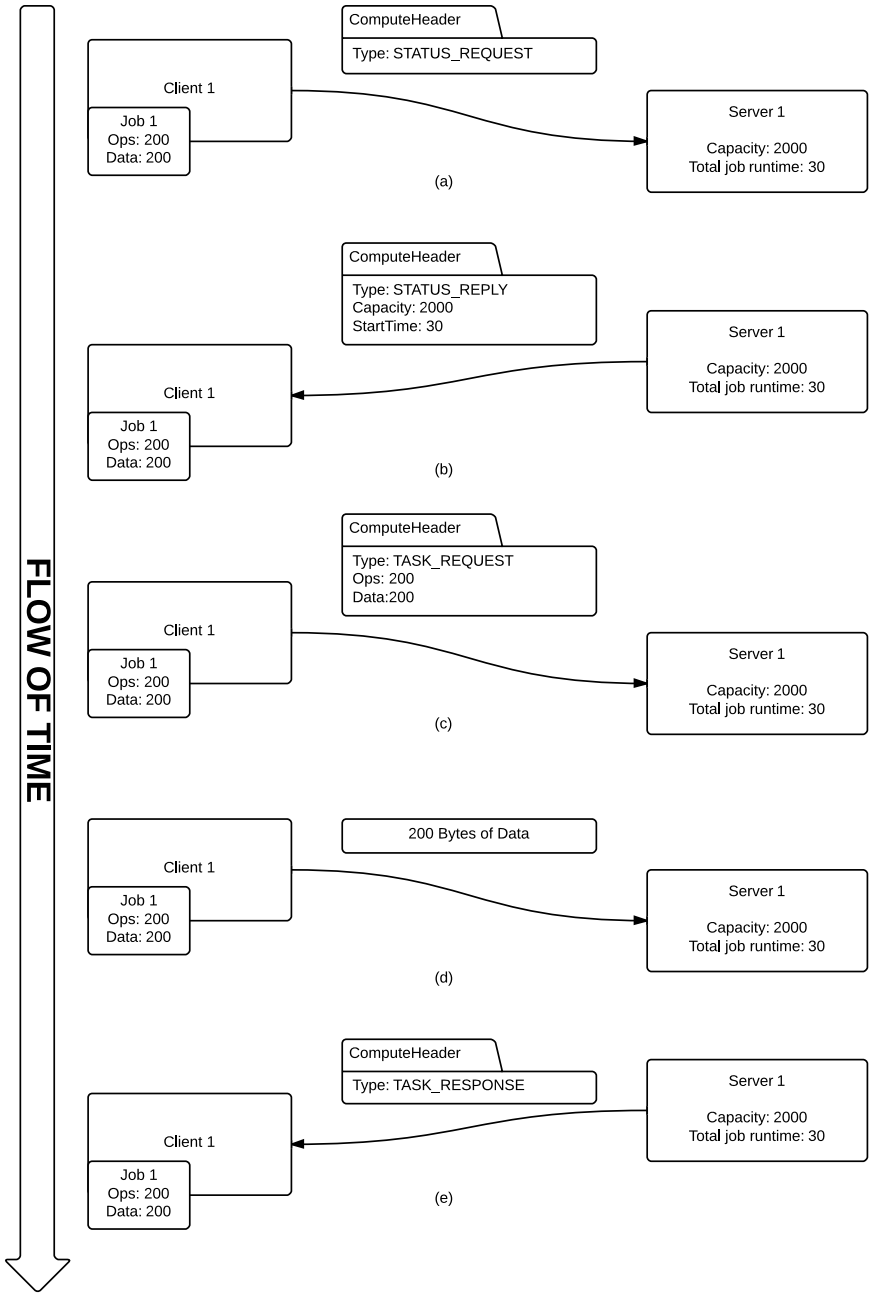

Figure 3. Example of typical communication between Client and Server.

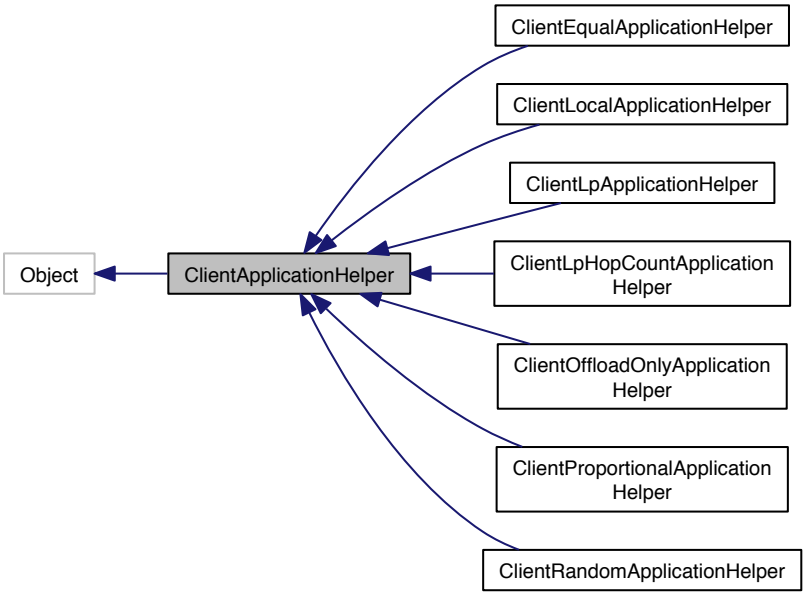

Figure 4. Application helpers for installing a Client Application onto an ns-3 node.

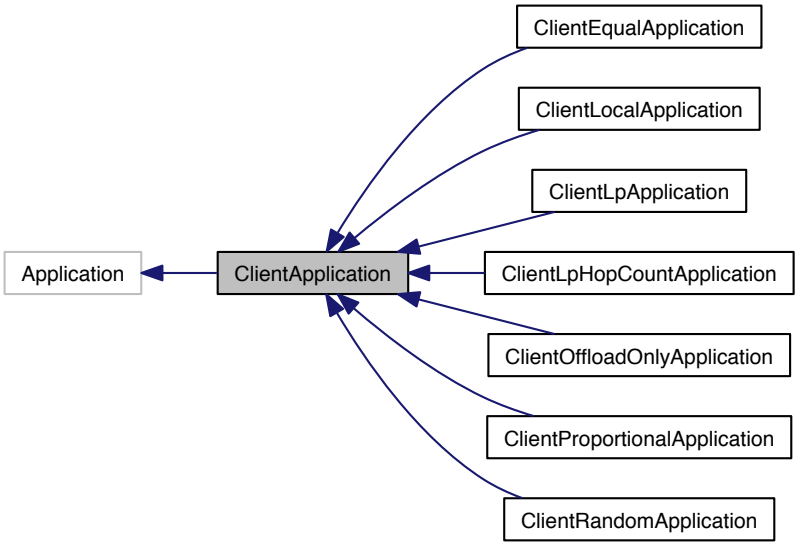

Figure 5. Different types of Client Applications. Each has a different offloading strategy.

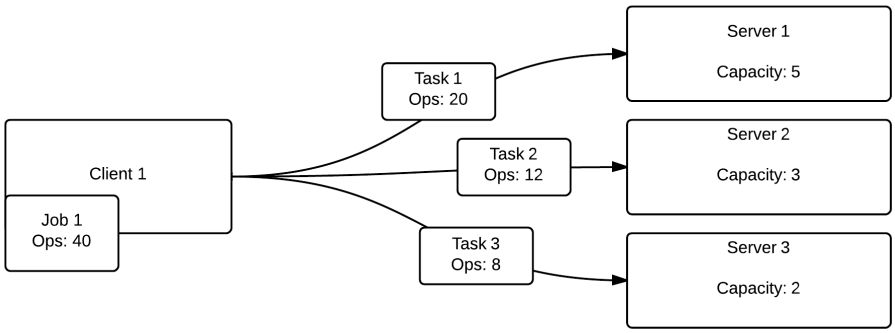

Figure 6. Example of typical communication between Client and Server.

```
void
ClientEqualApplication::DistributeJob()
{
   //Ensure at least one response has been received
   if(m_assetInfo.size() == 0)
   {
      //wait 2 more seconds and check again
      ns3::Simulator::Schedule(ns3::Seconds(2),
                  &ClientApplication::DistributeJob, this);
      return;
   }

   //Get job start time. This is used to calculate job completion time
   m_jobStarted = ns3::Simulator::Now();

   //Sort based on performance. Not all that useful here.
   std::sort(m_assetInfo.rbegin(), m_assetInfo.rend());

   //The number of responses that indicates job completion
   m_tasksExpectedReturned = m_assetInfo.size();


   //Fill out compute header so server has task information
   for(uint32_t i = 0; i < m_assetInfo.size(); ++i)
   {
     ComputeHeader header;
     header.SetHeaderType(ComputeHeader::TASK_REQUEST);
     header.SetSender(GetNode()->GetId());
     header.SetOps(m_jobSize / m_assetInfo.size();
     header.SetDataSize(m_dataSize);
     ns3::Ptr<ns3::Packet> packet = ns3::Create<ns3::Packet>();
     packet->AddHeader(header);

     m_serverSockets[m_assetInfo[i].nodeId]->Send(packet);

     //If data is being sent remotely, send it now
     if(m_assetInfo[i].nodeId != GetNode()->GetId())
       {
         ns3::Ptr<ns3::Packet> dataPacket =
                    ns3::Create<ns3::Packet>(m_dataSize);
         m_serverSockets[m_assetInfo[i].nodeId]->Send(dataPacket);
       }
   }
}
```

Figure 7. An Example Offloading Implementation.

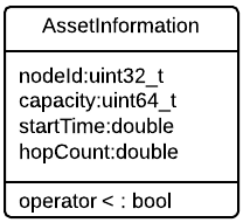

Figure 8. The client stores utilization information received from the server in the Asset Information class.

3.3.4 Finishing a Client Job

Once a client has made its offloading decision it waits to get responses from all servers to which it has sent a task. Once it has received job completion replies from each of these servers, it logs the total job time. The client then checks to see if it has more jobs to perform. If there are no more jobs to perform, the application ends. If there are more jobs to perform, the client again pauses for a randomly determined period. After the pause period the client resends utilization requests to all known servers and the process starts again.

3.4 Servers

This section discusses our model of an offload target. Offload servers have two responsibilities; they respond to utilization requests from clients and they process tasks assigned to them by clients. Like the ClientApplications, the server application inherits from ns3::Application, as shown in figure 9, and has an associated ServerApplicationHelper that is used to install it onto an ns-3 node. The attributes that the ServerApplicationHelper sets on the ServerApplication are listed in table 4. In this section we describe the operation of a server during a simulation.

For a utilization request the server returns two items, capacity and start time. Capacity is the number of operations per second that the server is able to compute—an analog to the actual hardware of a real-world device. This static value is set in the application by the application helper when the server application is created and is not changeable at run-time. The second value is start time. Unlike capacity, this value is dynamic and is used to represent how busy the server is. The value is calculated by traversing the server's task queue and summing the number of operations in each task to compute the total number of operations in the queue. The sum is divided by the server's capacity to get the total run time required to complete the server's task queue. For this value to be accurate, we must also add to this time the remaining run time for the task the server is currently running. This sum is returned to the client and represents the

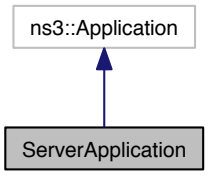

Figure 9. The Server Application.

Table 4. List of ns-3 Attributes in Server.

Attribute	ns-3 Type	Purpose
Local	AddressValue	The address (IP address and port number) that the listening socket on the server will bind to.
Protocol	TypeIdValue	The TypeId of the protocol to use (should be TCP).
Flops	UintegerValue	The number of operations the server can perform per second.

minimum amount of time that client's task will have to wait before it is executed on the server. It is important to note that this is a minimum. The server could receive other tasks before the client is able to transfer its task to the server, so the wait time may increase.

The second responsibility of a server is to perform tasks assigned to them by clients. A server is notified of an incoming task with the receipt of a ComputeHeader of type TaskRequest. From this the server knows the number of operations requested and the size of the data to receive. However, it may not yet have all of the data that has been sent. The server will not schedule the task until all of the data for the task is received.

The server processes tasks one at a time and in the order that they were placed onto the queue. Once a task has been completed, the server notifies the client via a ComputeHeader of type TaskResponse.

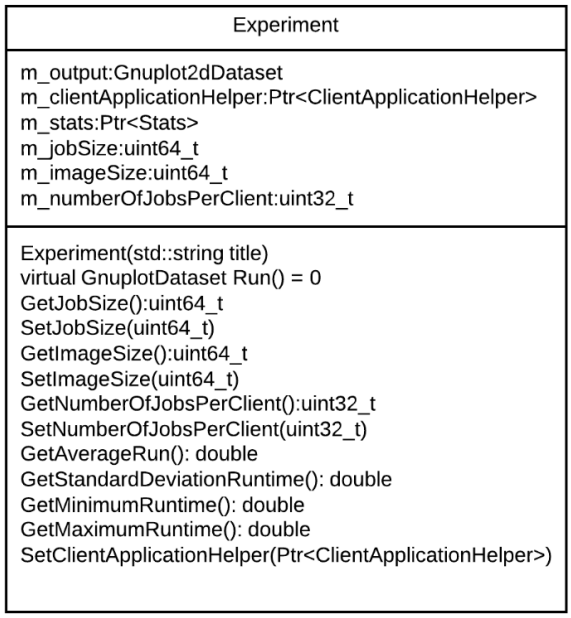

Figure 10. UML diagram of the Experiment Class.

3.5 Experiments

This section describes the Experiment class and its subclasses. The purpose of these classes is to make it easier for a user to create new — and modify existing—offloading experiments. The following section describes the layout and functionality of the Experiment class and subclasses. Individual experiment results using these classes will be presented in section 4..

Figure 10 shows the UML diagram for the Experiment class. This class is a base class for all experiments using the application framework. It is an abstract class; therefore, it cannot be instantiated directly. The SetClientApplicationHelper method sets the ClientApplicationHelper that is used to create the clients in the experiment. Also provided are methods for setting parameters for the experiment including number of jobs to run, job size, and image data size. Once the ClientApplication Helper has been set, modifying the number of jobs, job size, or data size in Experiment will modify the corresponding value in the ClientApplicationHelper. The virtual method Run must be implemented by any classes inheriting from the Experiment class. It returns a Gnuplot2dDataset containing experiment results that can be easily plotted. The class also provides methods for accessing statistical information obtained for the jobs performed by the clients including average run time, maximum and minimum run time, and standard deviation.

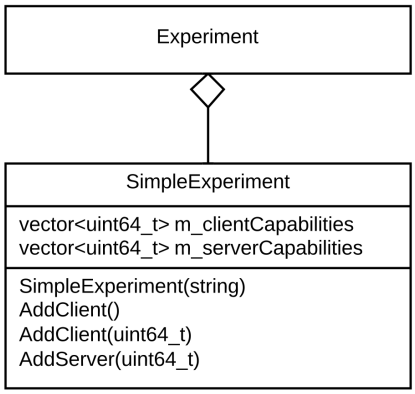

Figure 11. UML diagram of the SimpleExperiment Class.

3.5.1 Simple Experiments

SimpleExperiment is a child class of Experiment and is also an abstract class. See figure 11 for a UML diagram. This class offers an interface to add clients and servers to an experiment. Clients can be added with or without compute capability. The intent here is that clients without a compute capability will not have a ServerApplication installed on them and therefore, will always have to offload the entire job.

3.5.2 Simple Wired Experiments

ExperimentSimpleCsma and ExperimentSimplePointToPoint, shown in figure 12 are again abstract classes. These classes allow the testing of offloading strategies without the added complexity of wireless ad-hoc networks. The purpose of these classes is to wrap ns-3 specific code in configurable and reusable classes.

Each of these classes has a RunIndividual method, which contains all of the ns-3 code for the experiment. This method sets up the network topology, configures all of the network devices, and installs client and server applications on the appropriate nodes. Client nodes that were added to the experiment without a capacity receive only a client application. Client nodes that were added with a capacity receive a client application and a server application with the specified capacity. Server nodes are installed with a server application with the given capacity. The type of client application installed is determined by the ClientApplicationHelper given to the experiment through the interface provided by the base class Experiment. If no application helper is given, it defaults to ClientLocalApplication (i.e., no offloading).

In ExperimentSimplePointToPoint all of the links between nodes are individual wired

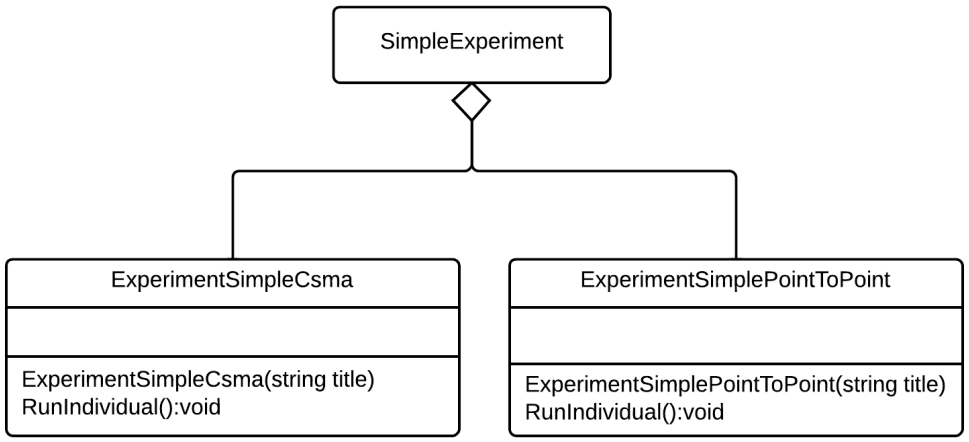

Figure 12. UML diagram of ExperimentSimpleCsma and ExperimentSimplePointToPoint.

point-to-point links. Each client in the experiment is connected directly to each of the available servers. In the standard configuration there are three servers and the client sends to each at a different data rate. The data rates, or the rate at which the network device places data on the wire, that the clients use are 54 Mbps, 18 Mbps, and 6 Mbps—which are common Orthogonal frequency-division multiplexing (OFDM) data rates. figure 13 shows an example of the type of topology this class creates. In the topology, both clients send to server 1 at a rate of 54 Mbps, server 2 at 18 Mbps, and server 3 at 6 Mbps. In the experiment, the number of servers must be a factor of three because the rates repeat after the third server. For example with six servers, the clients would send to both server 1 and server 4 at a rate of 54 Mbps.

ExperimentSimpleCsma is similar to ExperimentSimplePointToPoint with the exception of the network devices used by the nodes and the links between them. In ns-3, Carrier sense multiple access (CSMA) is a simple bus network similar to Ethernet (IEEE 802.3). With this configuration, the server and all of the clients that connect to it share one bus network. With this configuration there will be delays when multiple clients attempt to send data to the same server at the same time. This differs from the point-to-point network where each client had an individual link to each server.

ExperimentSimplePointToPoint and ExperimentSimpleCsma are used by calling the RunIndividual method. Typically, this is done from within the Run method, which implements the purely virtual Run method in the base Experiment class. An example of a class that does this is ExperimentSimplePointToPointVaryDataSize, figure 14. This experiment measures the effect

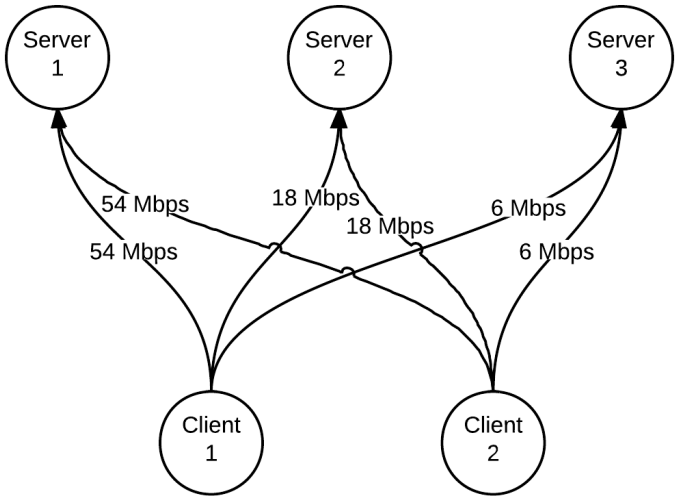

Figure 13. The Experiment Simple Point to Point Vary Image Size class runs an experiment varying the image size, or amount of data, each server receives along with the task request.

of changing the data size sent to the servers during the offloading process. Figure 15 shows the code for the experiment. The experiment measures 21 different image sizes. For each size, the average client run time is stored in a dataset.

3.5.3 Simple Wireless Ad-Hoc Experiments

ExperimentSimpleWireless Ad-Hoc is similar to ExperimentSimplePointToPoint and ExperimentSimpleCsma in that it wraps ns-3 specific code. However, it differs in that it models an IEEE 802.11 wireless network in ad-hoc mode. For routing, this experiment uses the OLSR Protocol. Being that this is a wireless topology, the position of the nodes is now important and this experiment contains two ns-3 position allocators: one for clients, one for servers. All units for ns-3 position allocators are in meters. By default, the client position randomly places client nodes. The server position allocator implements no default and must be set manually.

The ns-3 has multiple wireless channel propagation loss models. ExperimentSimpleWirelessAd-Hoc uses the RangePropagationLossModel. In this model, every node within a radial distance of the transmitter will receive the signal at full power. We chose this model because it is difficult to accurately model wireless propagation loss unless the exact environmental topology is known. See (*10*) for a discussion of the propagation loss models available in ns-3.

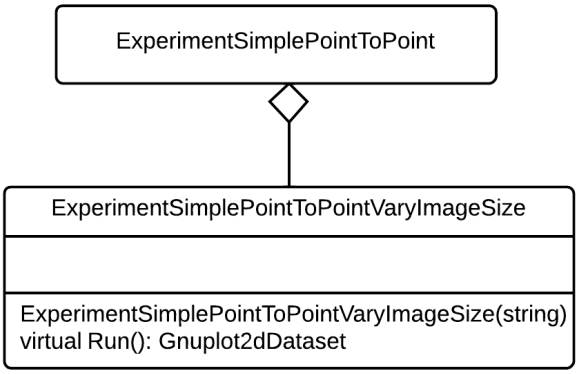

Figure 14. The ExperimentSimplePointToPointVaryImageSize class runs an experiment varying the image size, or amount of data, each server receives along with the task request.

3.5.4 Multiprocess Experiments

Since ns-3 is a single-threaded process, it does not take advantage of the multiple cores commonly found on current CPUs. Therefore, we have created an experiment that spawns multiple experiments each running a single-threaded ns-3 instance. This functionality is controlled by the ExperimentMultiprogramDriver class as shown in figure 17. This abstract class spawns multiple subprocesses and passes experiment information to them using named pipes.

Client and server information for this type of experiment are stored in the Asset class as shown in figure 18. This class can easily be serialized via the named pipe to the subprocess. Again, the distance specified by the coordinates in this class is in meters.

For each subprocess, the information for the clients is the same. Clients will be in the same location for each subprocess experiment and have the same compute ability. For servers, this is not the case. Each item in the outer vector of m_servers represents one subprocess experiment. The inner vector specifies the server locations and capabilities for that particular subprocess experiment.

When RunMultiple is called, the main process will spawn N subprocesses, where N is the number of items in the outer vector of m_servers. Along with the information regarding the clients and servers, the main process passes the job size in operations, the image size in bytes, and the number of jobs for each client to the subprocesses over the named pipe. These values are the same for each subprocess experiment.

```
ns3::Gnuplot2dDataset
ExperimentSimplePointToPointVaryDataSize::Run()
{
   //Add 3 servers with varying compute capacity
   this->AddServer(4e9);
   this->AddServer(6e9);
   this->AddServer(12e9);

   //Add 3 clients
   this->AddClient();
   this->AddClient();
   this->AddClient();

   for(uint32_t i = 0; i <= 20; ++i)
   {
      //reset stats
      m_stats->jobCompletionTime.clear();

      //vary between 20 and 200 MB
      uint64_t dataSize = 2e6 + (i * 9900000);

      NS_LOG_INFO("Running dataSize " << dataSize);

      //set the image size
      this->SetDataSize(dataSize);
      //Could have also done this
      //m_clientApplicationHelper->SetAttribute("DataSize",
      // ns3::UintegerValue(dataSize));

      //run experiment
      RunIndividual();

      //get average run time
      double averageRunTime = GetAverageRuntime();

      //add average RT to dataset
      m_output.Add((double)dataSize, averageRunTime);
   }
   return m_output;
}
```

Figure 15. An example experiment.

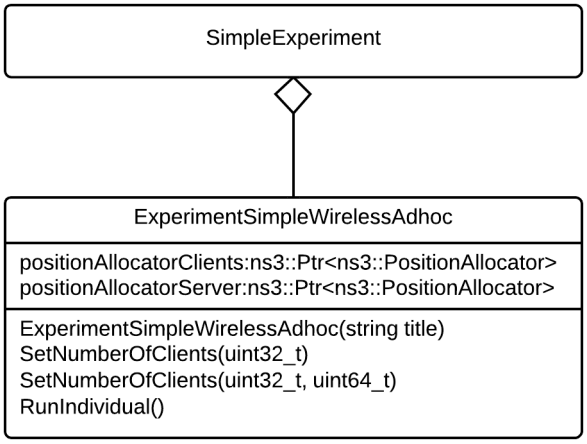

Figure 16. UML diagram of ExperimentSimpleWirelessAdhocUML Class.

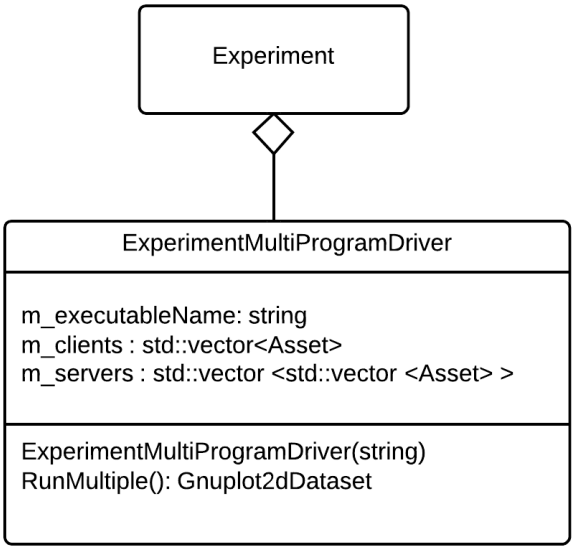

Figure 17. UML diagram of ExperimentMultiProgramDriver Class.

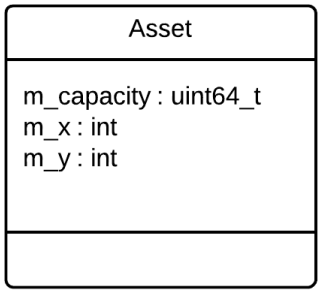

Figure 18. The Asset class stores Clients and Servers in a format that can be easily serialized.

When the ExperimentMultiProgramDriver class calls a subprocess, it attempts to execute the executable from which it is being run. Therefore, when using the multiprocess experiment, the main program (the file with the method main in it) must be configured correctly so that it can be called as a subprocess. Each subprocess is assigned an experiment number and this number is passed as an argument to the subprocess when it is executed. Therefore, the main method can check for the presence of this argument, and if it exists it should run as a subprocess. Otherwise it should run as the main process and use an instance of ExperimentMultiProgramDriver.

ExperimentServerPlacement is currently the only experiment that can be run on a subprocess. It inherits from ExperimentSimpleWirelessAd-Hoc, as shown in figure 19 and uses a wireless ad-hoc topology. When the subprocess is run, it is passed an experiment number. This is also the name of the named pipe from which the experiment will get its client and server information. Prior to running the experiment it is necessary to call LoadInformationFromFifo with the appropriate experiment number. At this point the experiment can be run. Once the experiment is complete, the subprocess reopens the named pipe that it received its information on and writes out the average runtime for all clients during the simulation. This class also outputs a file of statistics including the minimum, maximum, and the standard deviation for client run times. The file is generated in the folder where the main program was originally run. The statistics are listed by the experiment number. We have written a simple python program to transfer these experiment numbers to x and y coordinates.

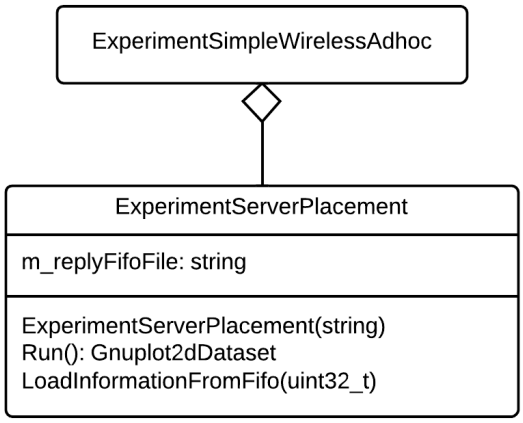

Figure 19. The ExperimentServerPlacement class is used in a subprocess to run an ad-hoc experiment.

4. Experiments and Results

In this section we present results obtained using the application models and experiment classes described in section 3.

4.1 Offloading on Wired Networks

In this set of experiments, we examine the different offloading strategies on a wired network. The purpose of these experiments is to test the initial offloading models in a controlled environment without the added complexity of wireless transfers over an ad-hoc network. Specifically, we wanted to test how our newly proposed utilization aware offloading strategy described in section 3.3.3 performed. These initial results were first presented in a previous work (*8*).

In these experiments, we use the topology shown in figure 20. In the figure, the computation capability of each of the servers is shown above each node. The computation capability of two GeForce GTX Titan gaming Graphics processing units (GPUs), currently some of best hardware available, were used for highest capability offloading target. This value was scaled down for the medium and low capability targets. The client devices were given computation capabilities similar to that of current high-end smart phones, such as the iPhone 5 (*11*). These are around three orders of magnitude less than the computation capabilities of the servers. The data rate for the links between the client and servers are standard bandwidth values used by OFDM. The connections in this topology represent a worst-case scenario in that the fastest link is connected to the slowest server while the slowest link is connected to the fastest server. We did this to demonstrate the importance of considering the entire network state when making an offloading decision.

4.1.1 Varying Data Size

In the first experiment, we vary the data size, or the amount of data that needed to be transferred to complete the job. This allowed us to evaluate the performance of offloading data-intensive jobs. We varied this value from 2×10^7 bytes to 2×10^8 bytes. For this experiment, the job size in operations is directly proportional to the number of bytes transferred. The R value, as described in section 3.3.3, was 72,000 operations per byte. Therefore, as the transmission size increased, so did the job size. For this experiment we compared four different offloading strategies: local, architectural-aware, architectural and utilization aware, and equal distribution. figure 21 shows the results of this experiment. As expected, for each offloading strategy, as the data size and job

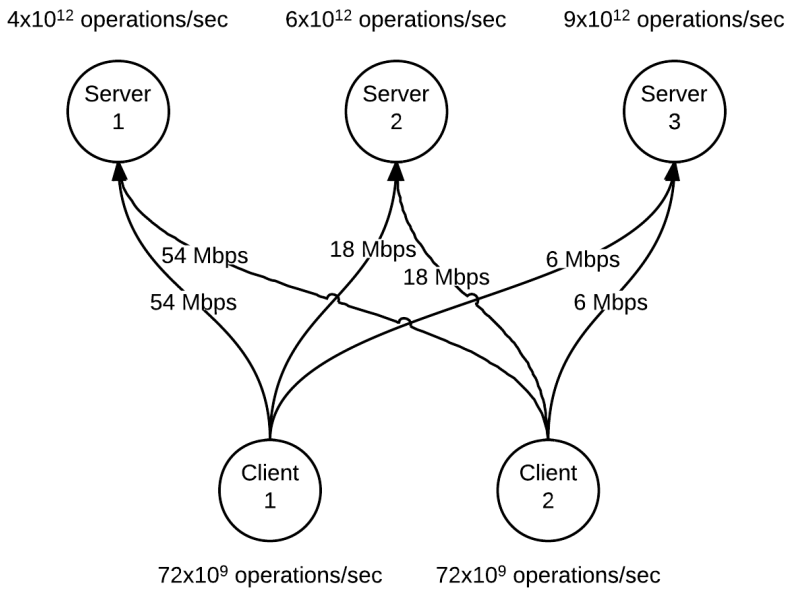

Figure 20. The network topology used for our wired offloading experiments. The compute capacity of each device is located above or below it.

size was increased, the average run time also increased. The Utilization aware algorithm has the lowest average run time. Architectural-aware offloading performed worse than just simple equal distribution, which was unexpected. The reason for this is that the bandwidths of the links are inversely proportional to the capabilities of the servers. Therefore, the architecture-aware model was attempting to send a large amount of data over the slowest link. This demonstrates why it is important to consider the entire network state when making offloading decisions.

4.1.2 Varying Job Size

In the second experiment of this set we varied the job size (number of operations), by adjusting R, while keeping the image size fixed at a moderate 2 MB. We did this to evaluate how the algorithms would perform with computationally intensive jobs. Specifically the job size was varied from 2×10^{12} operations to 1.8×10^{13} operations. Figure 22 shows the results of these experiments. Note that the local and equal strategies are not shown here because the average runtimes were much higher. This result was expected since the computational ability of the device is magnitudes less than the servers. Equal distribution also performs poorly here because one piece of the job is performed locally on the comparatively slow handheld device. As the results show, the utilization-aware algorithm performs better than the architecture-aware

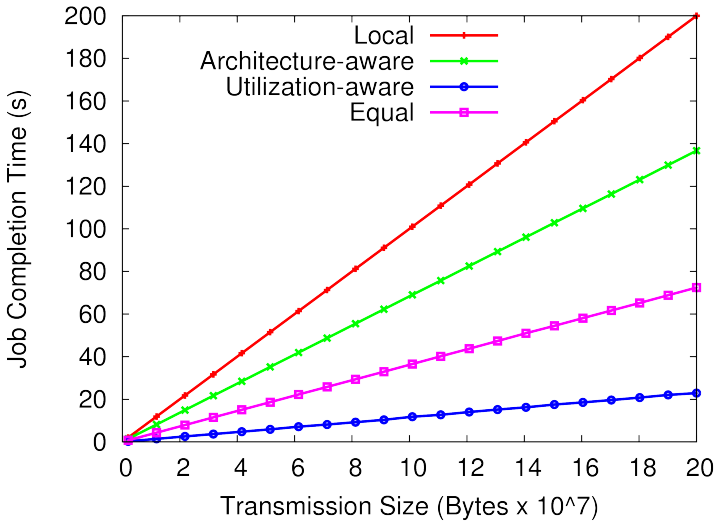

Figure 21. As the job size and data size increase, the time to complete the job increases linearly. The job completion time for equal distribution increases at a rate of 3.6 s/B, while the utilization-aware offloading algorithm has a job completion time increase of only 1.1 s/B.

algorithm on all job sizes.

4.1.3 Varying Number of Clients

The next experiment was to increase the number of clients offloading jobs. In this experiment we varied the number of clients from 1 to 10. The job size was set small enough so that the equal algorithm was to perform comparatively to the architecture- and utilization-aware algorithms. The results of this experiment are shown in figure 23. Again the utilization aware algorithm has the best performance because it is taking into account the load on the server as well its computational capacity. The architecture aware algorithm continues to send the same amount of each job to each server regardless of its load. This results in a large task queue on the fastest server as the number of clients increase resulting in slower average run times.

4.1.4 Varying Number of Offload Targets

In the last experiment in this set we examined how the algorithms performed as the number of offloading targets increased. The number of servers in the experiment was increased from the original 3 to 21 by increments of 3. The three servers added at each iteration have the same characteristics as the original three servers. The results of this experiment are shown in figure 24. We see that the equal distribution algorithm has the greatest benefit from the addition of more servers. This is because the size of the task that the equal distribution algorithm assigns the local

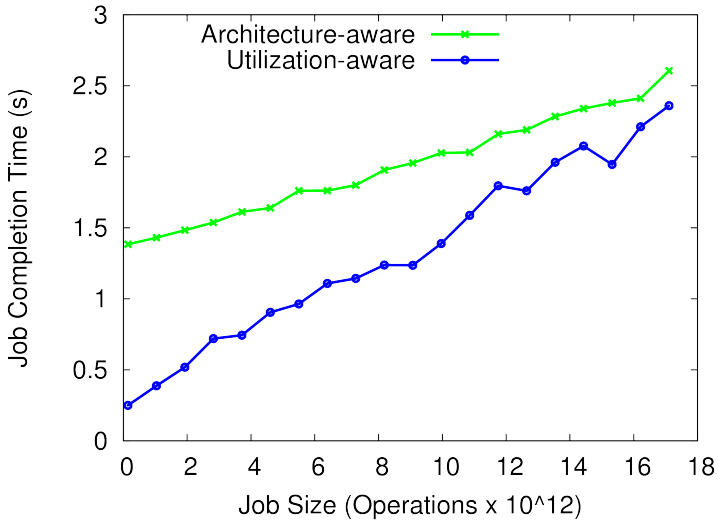

Figure 22. The performance of offloading algorithms varying the job size. Not shown are local execution and equal distribution because they took orders of magnitude longer to complete than architecture-aware and utilization-aware.

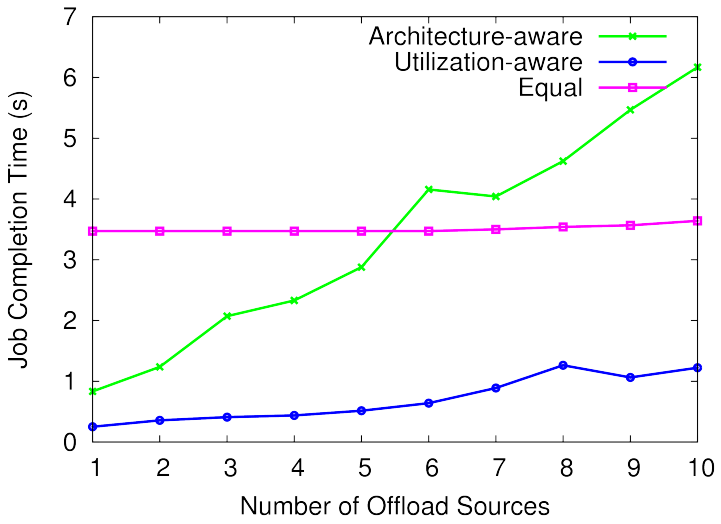

Figure 23. Average job completion times varying the number of clients. The utlization-aware algorithm outperforms the alternatives.

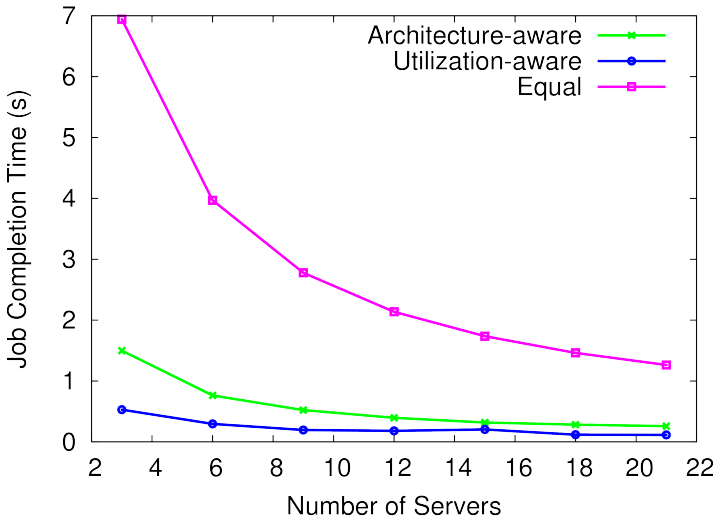

Figure 24. Increasing the number of servers allows greater parallelism as a job can be distributed across more devices.

server decreases as the number of servers increase.

4.2 Effects of Offload Target Location

In the next set of experiments we examined how positioning of the mobile HPC affected the job completion time. The experiment consisted of multiple ns-3 runs each with a single server placed in a different location. For these experiments we used the ExperimentSimpleWirelessAd-Hoc experiment model. All of the experiments in this section used the architecture-aware offloading strategy since only one server was being used. The maximum range used for the Range Propagation Loss Model on each node was 155 meters. The wireless transmitters were configured to transfer data at a constant 54 Mbps. Each client was given two jobs to run. The job size was set to 1×10^{12} operations. The sole server was given a compute capacity of 4×10^{12} operations per second. Therefore, the run time for the server to complete a job .25 s. All of the clients were given a compute capacity of 8×10^{9}. Therefore if the job was to be performed locally it would take 125 s.

The experiments were performed on a 960 square-meter grid. The server was initially placed at coordinate (-480, -480). Each subsequent run increased the y coordinate by 60 m until it reached 480 m, at which point the x coordinate was increased by 60 m and the y coordinate was reset to -480 m. This was repeated until the server reached (480, 480); a total of 289 (17×17) runs were performed per experiment. The ExperimentMultiProgramDriver class was used to launch the individual runs.

Table 5. Client positions for 5 Node Surface Experiment.

Client Node	X Position (m)	Y Position (m)
0	0	0
1	0	-145
2	0	145
3	-145	0
4	145	0

The following sections displays the results of running the experiment varying the number of clients and varying the data size sent to the server. The data will be presented in two forms. The first will be a scatter plot showing the average run time for the clients while the server is placed at that position. The size of the dot and the color represent the average run time. Large blue dots on the perimeter are locations where the server is out of range of all of the clients and computation is performed locally. The second form is a histogram showing when the individual jobs finished. The results are separated by the number of wireless hops between the client and the server when the client sends the job. In some cases where job times varied over a long period of time the histogram is split into multiple graphs. At the end of the results we offer a discussion.

4.2.1 Five Client Ad-Hoc Experiments

For the first set of experiments, five clients were used. Their positions are shown in table 5. All coordinates are in meters and are the distance from the corresponding axis on the grid. We ran the five client configuration with three different offloading data sizes: 1 KB, 100 KB, and 1 MB.

Figure 25 shows how the average run time varies as the server is moved through the clients. Figure 25 gives the histogram for the individual job completion times.

Next we increased the data offload size by 100 $\times$ to 100 KB and reran the experiment. The results for the average job run time at each location can be seen in figure 27. Figure 28 gives a histogram of job run times.

We next increased the data offload size to 1 MB. The results for average job run time at each location can be seen in figure 29. The histogram is shown in figure 30.

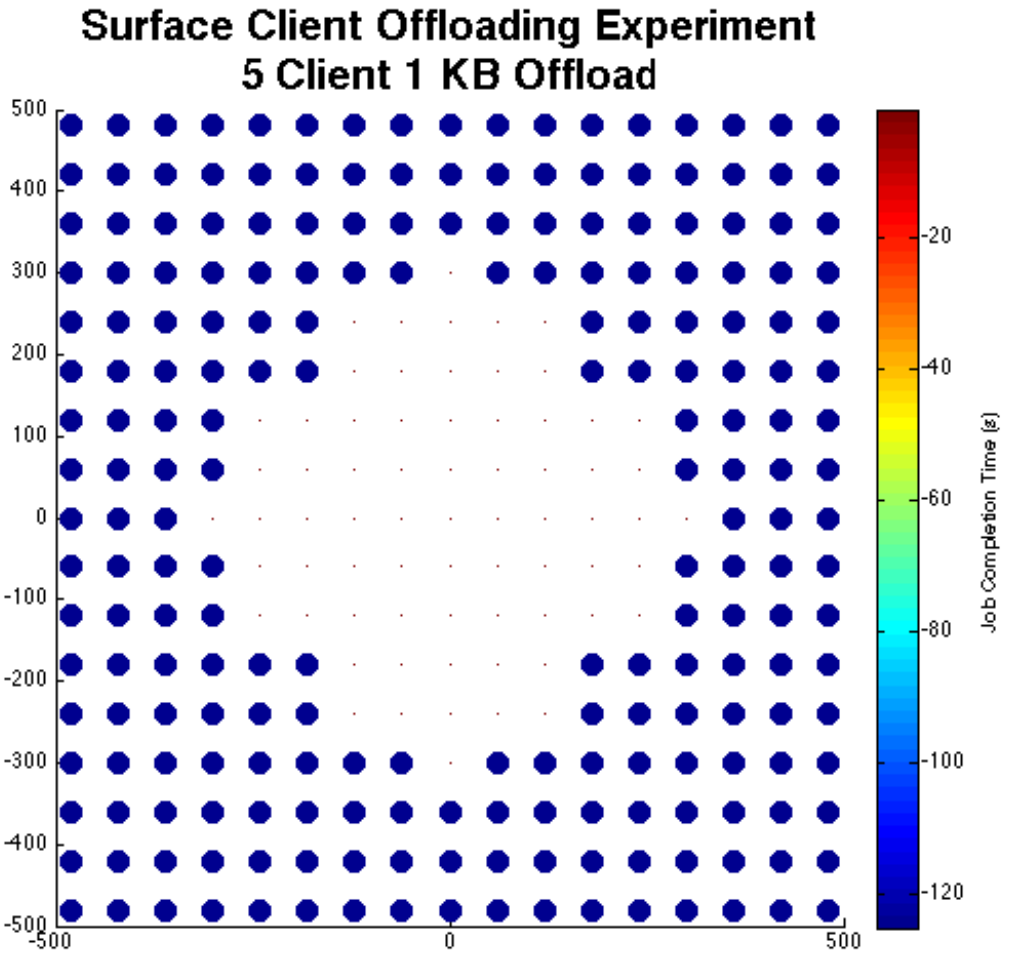

Figure 25. The histogram shows the number of jobs finishing between 0 and .7 s for the 5 client, 1 KB offload experiment.

Figure 26. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

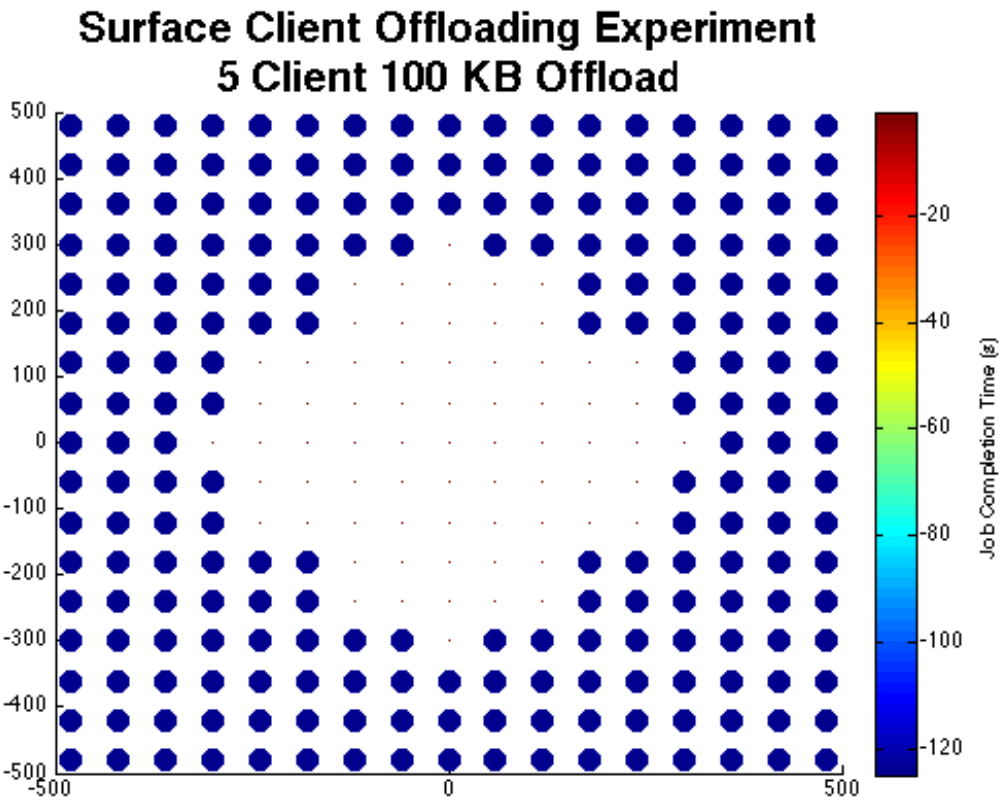

Figure 27. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

Figure 28. The histogram shows the number of jobs finishing between 0 and 2.1 s for the 5 client, 100 KB offload experiment.

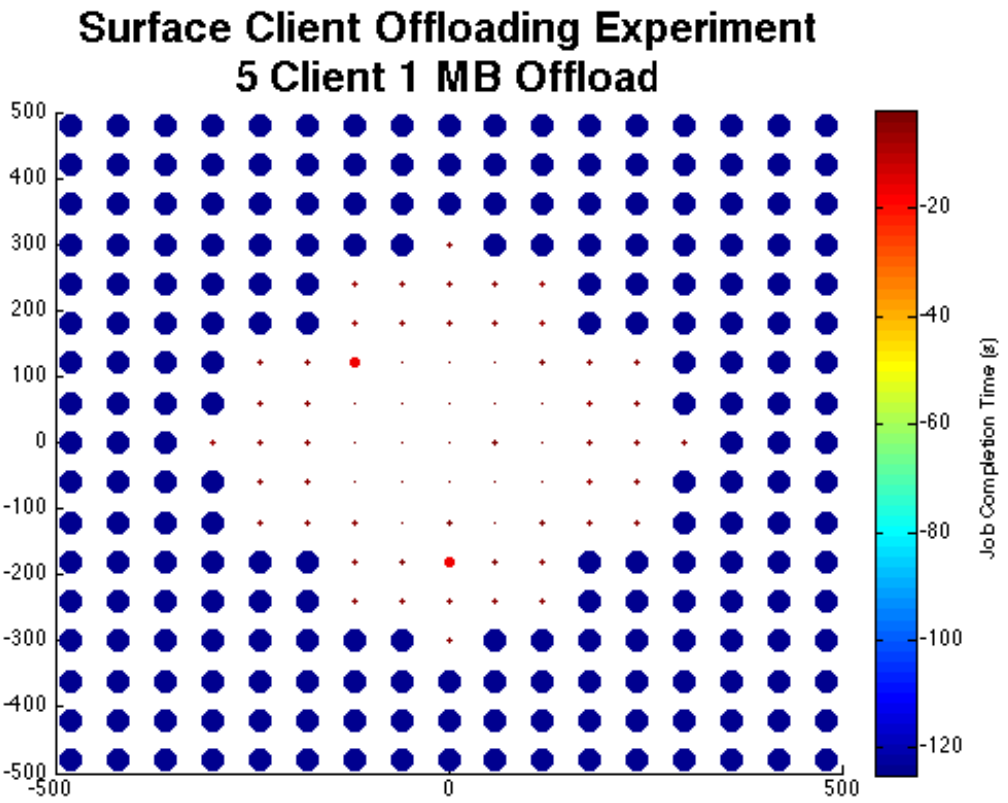

Figure 29. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

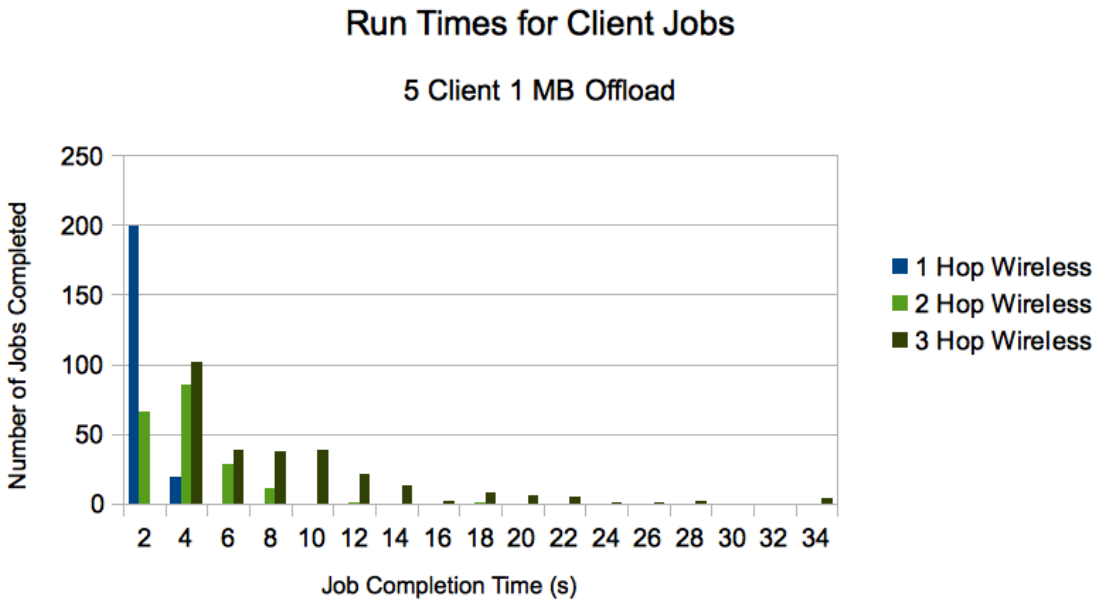

Figure 30. The histogram shows the number of jobs finishing between 0 and 34 s for the 5 client, 1 MB offload experiment.

Table 6. Client positions for 9 Node Surface Experiment.

Client Node	X Position (m)	Y Position (m)
0	0	0
1	0	-145
2	0	145
3	-145	0
4	145	0
5	-102	-102
6	102	-102
7	102	102

4.2.2 Nine Client Ad-Hoc Experiments

For the second set of experiments, nine clients were used. Their positions are shown in table 6. All coordinates are in meters and are the distance from the corresponding axis on the grid. Again we used three different data offload sizes: 1 KB, 100 KB, and 1 MB.

The average run times for the nine client, 1 KB experiment are shown in figure 31. The histogram is shown in figure 32.

We next increased the data offload size to 100 KB. Figure 33 shows the average run times for this configuration. The histogram for this setup is shown in figure 34.

Finally we increased the data offload size to 1 MB. Figure 35 shows the average run times for the server positions. Figure 36 shows the histogram of client completion times for the first 36 s. Figure 37 gives a histogram of the remaining jobs.

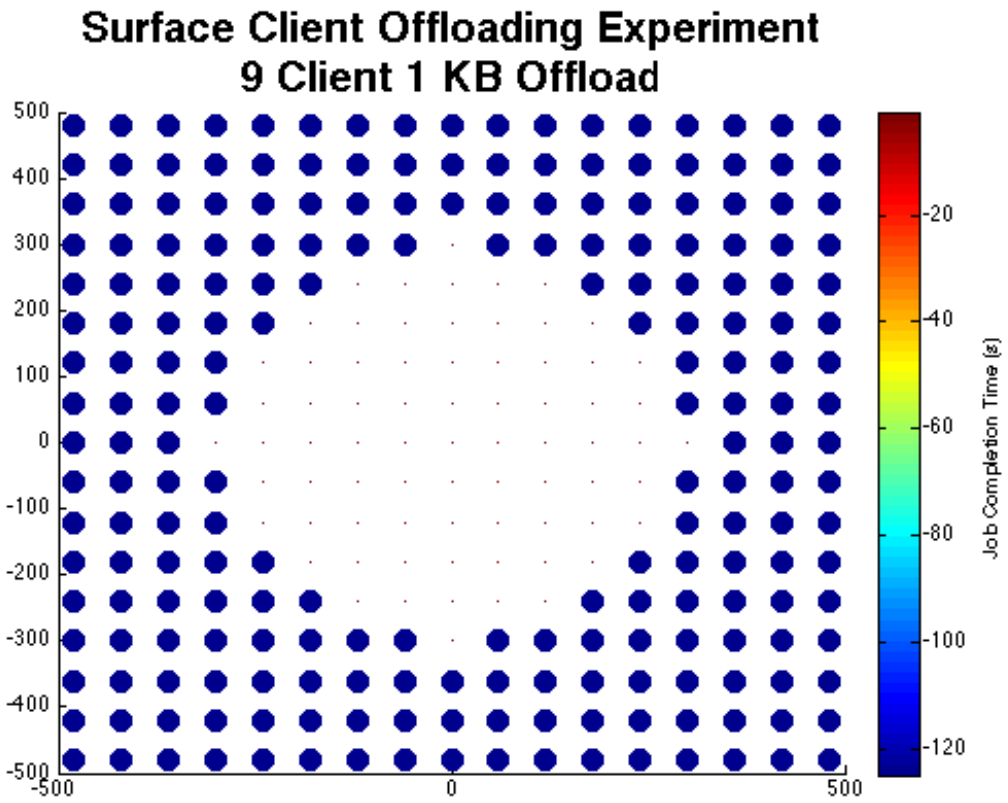

Figure 31. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

Figure 32. The histogram shows the number of jobs finishing between 1 and 1.5 s for the 9 client, 1 KB offload experiment.

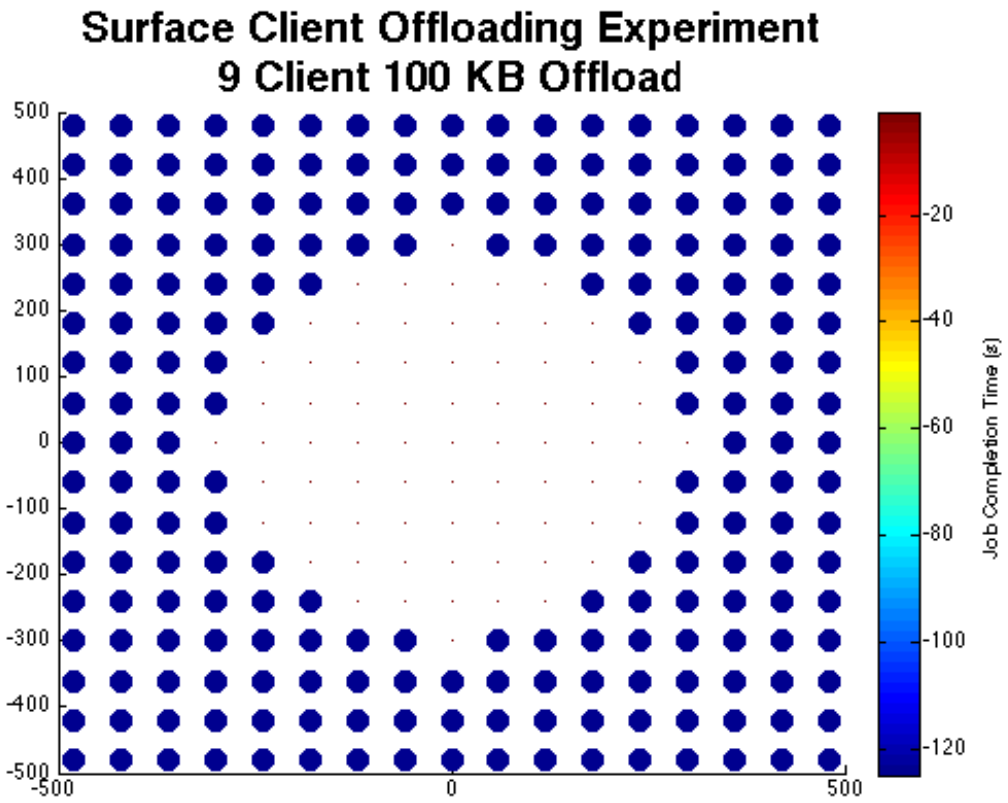

Figure 33. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

Figure 34. The histogram shows the number of jobs finishing between 0 and 2.8 s for the 9 client, 100 KB offload experiment.

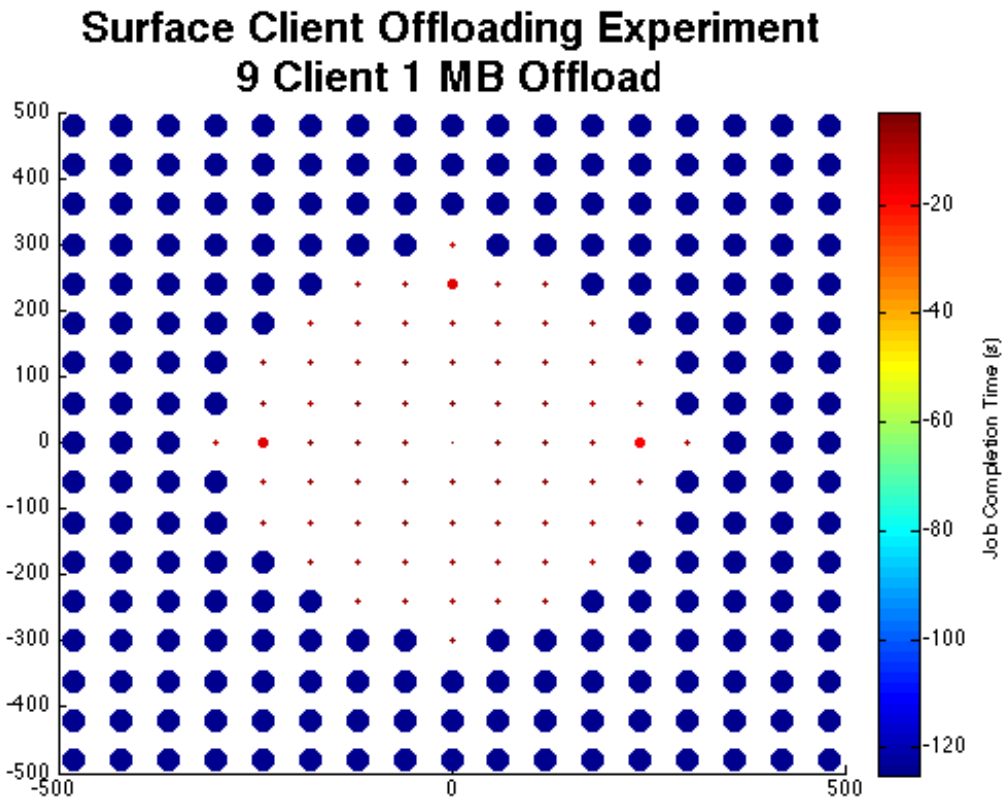

Figure 35. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

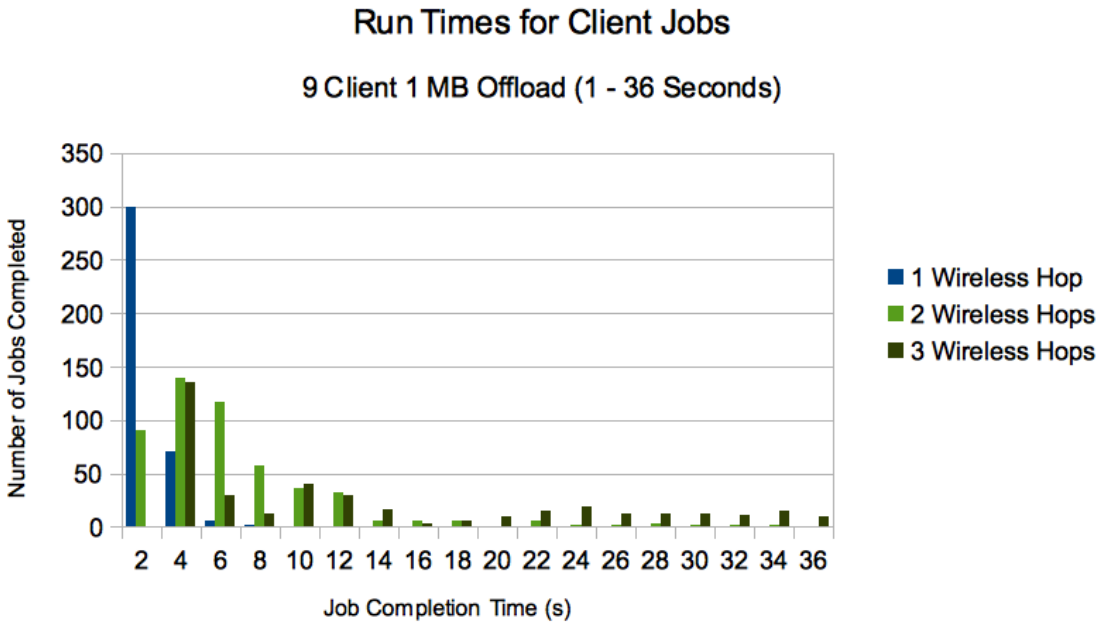

Figure 36. The histogram shows the number of jobs finishing between 0 and 36 s for the 9 client, 1 MB offload experiment.

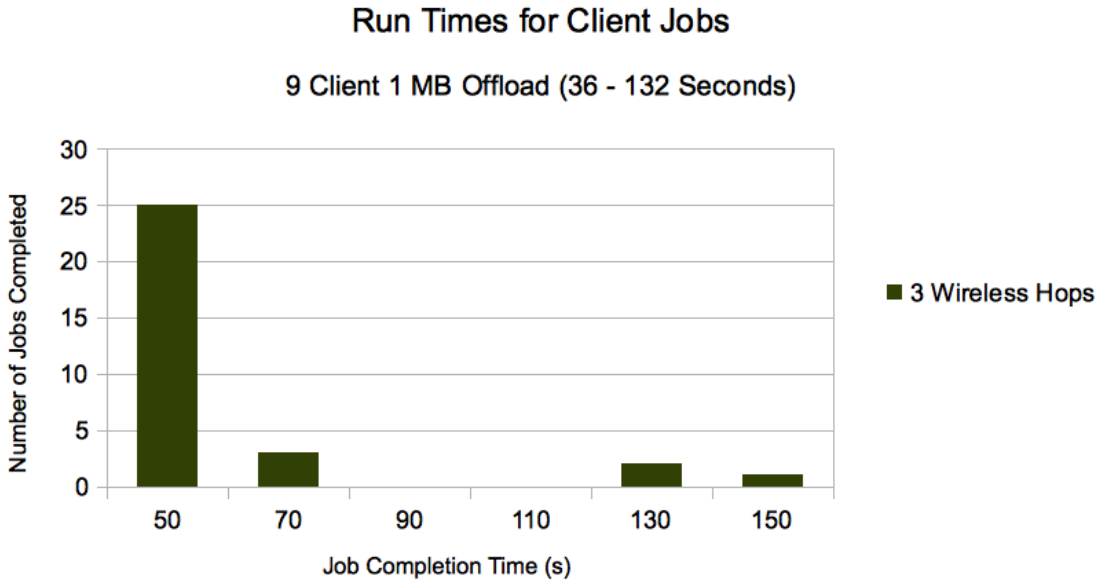

Figure 37. The histogram shows the number of jobs finishing between 36 and 132 s for the 9 client, 1 MB offload experiment. Note the change in scale on both axis.

Table 7. Client positions for 21 Node Surface Experiment.

Client Node	X Position (m)	Y Position (m)
0	0	0
1	0	-145
2	0	145
3	-145	0
4	145	0
5	-102	-102
6	102	-102
7	102	102
8	-102	102
9	0	-300
10	0	300
11	-300	0
12	300	0
13	-204	-204
14	204	-204
15	204	204
16	-204	204
17	0	204
18	0	-204
19	204	0
20	-204	0

4.2.3 Twenty-One Client Ad-Hoc Experiments

For the third set of experiments, 21 clients were used. Their positions are shown in table 7. All coordinates are in meters and are the distance from the corresponding axis on the grid. Again we ran the 21 client configuration with three different offloading data sizes: 1 KB, 100 KB, and 1 MB.

The average run times for the 21 client, 1 KB experiment are shown in figure 38. The histogram is shown in figure 39.

We next increased the data offload size to 100 KB. Figure 40 shows the average run times for this configuration. The histogram for jobs completing in the first 8 s is shown in figure 34. The remaining jobs are represented in figure 42.

Finally, we increased the data offload size to 1 MB. Figure 43 shows the average run times for this configuration. The histogram for jobs completing in the first 10 s is shown in figure 44. The

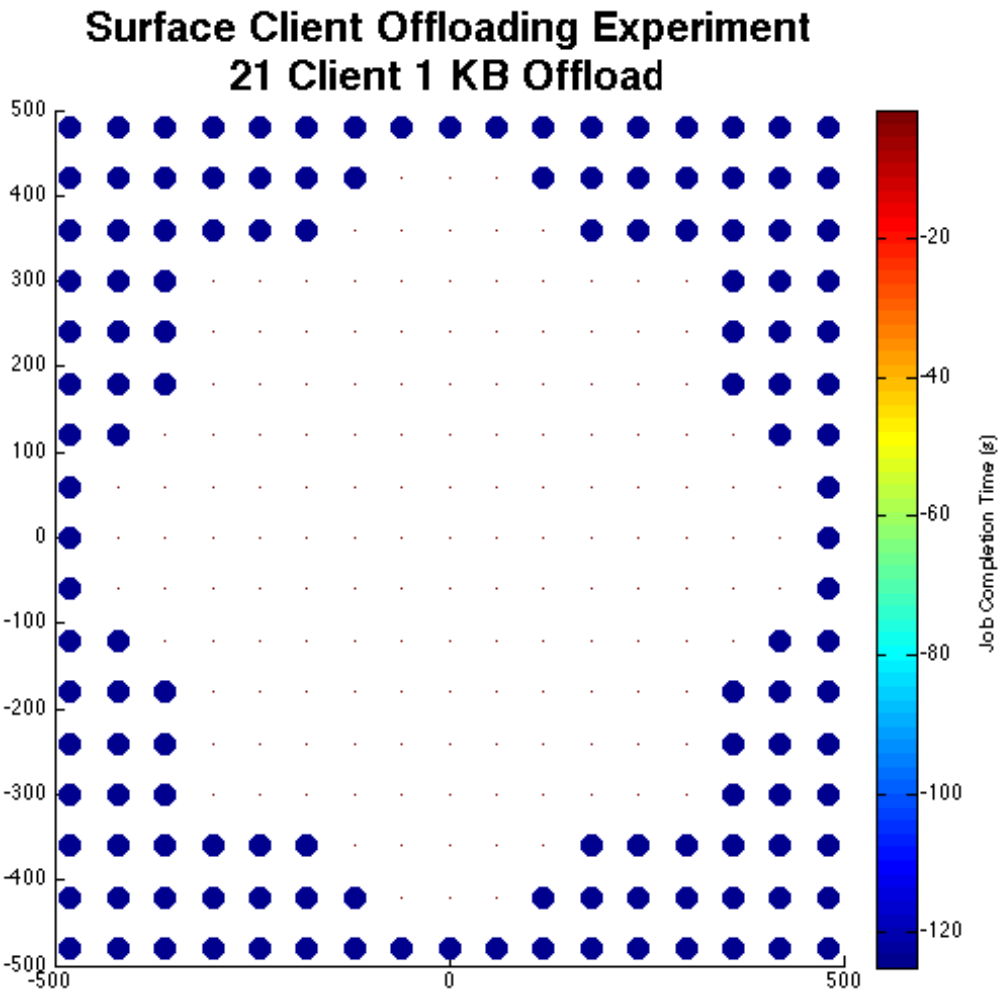

Figure 38. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

Figure 39. The histogram shows the number of jobs finishing between 0 and 5 s for the 21 client, 1 KB offload experiment.

remaining jobs are represented in figure 45.

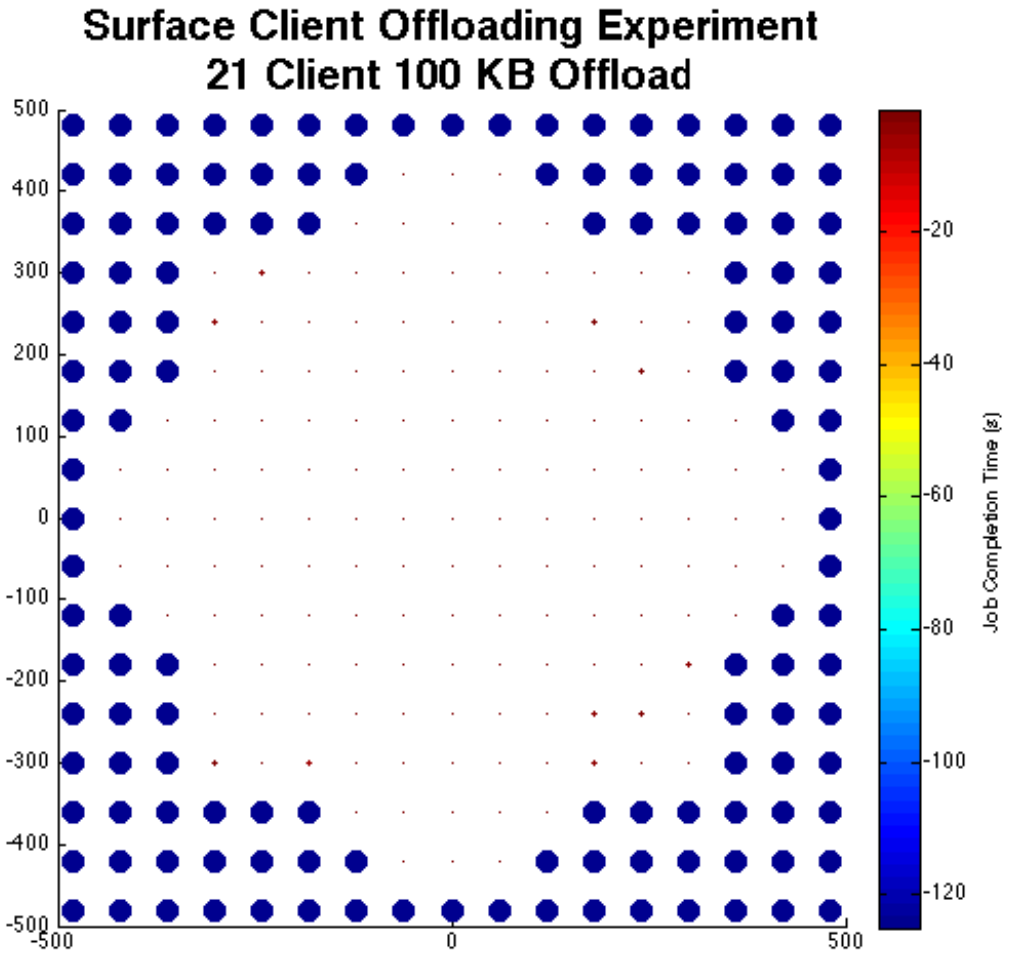

Figure 40. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

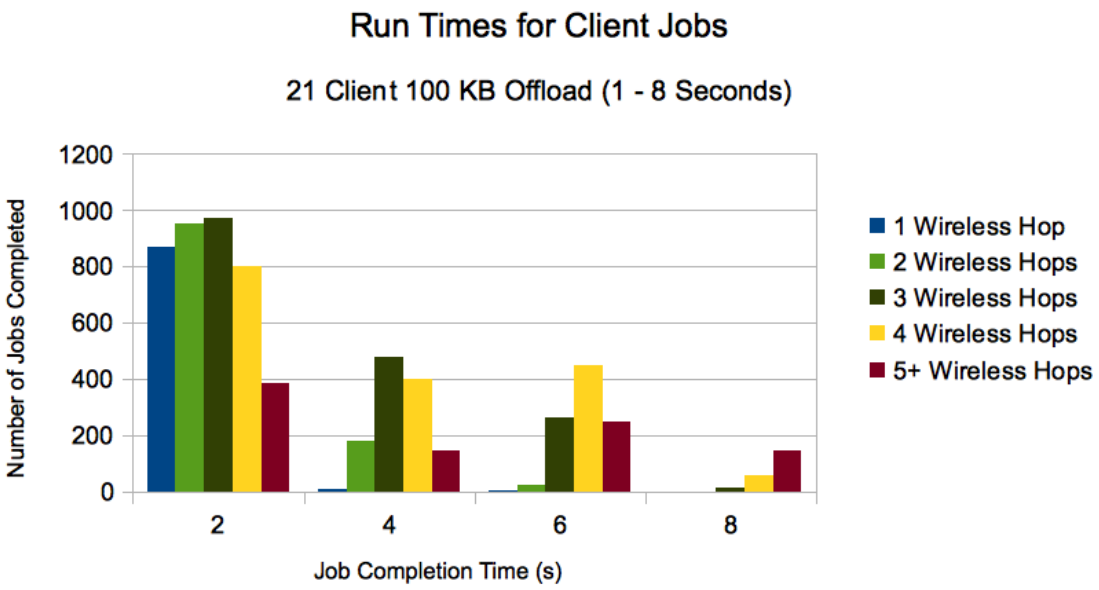

Figure 41. The histogram shows the number of jobs finishing between 0 and 8 s for the 21 client, 100 KB offload experiment.

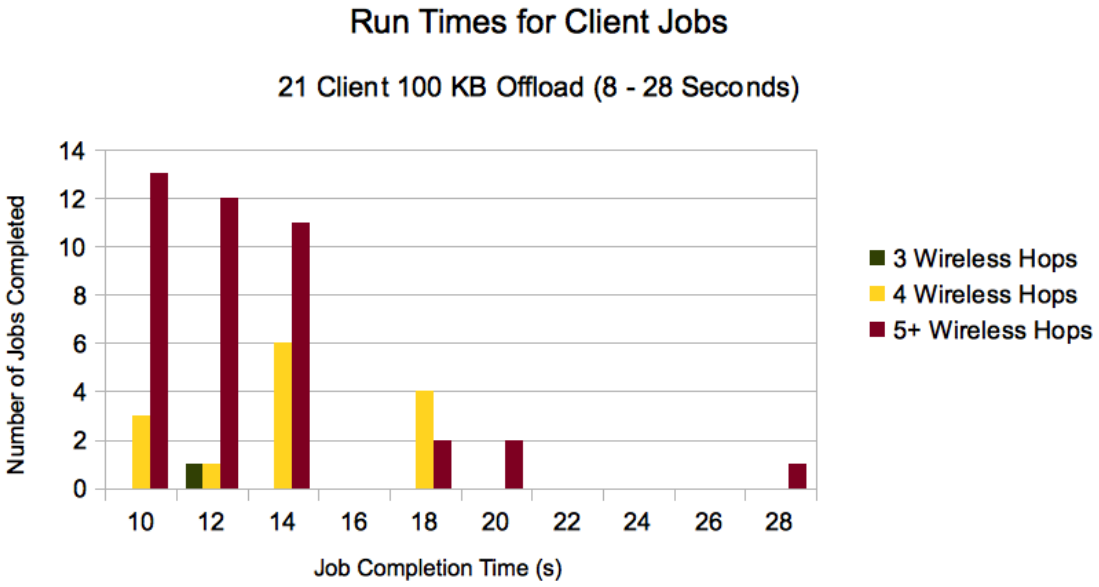

Figure 42. The histogram shows the number of jobs finishing between 8 and 28 s for the 21 client, 100 KB offload experiment. Note the difference in scale on the y axis.

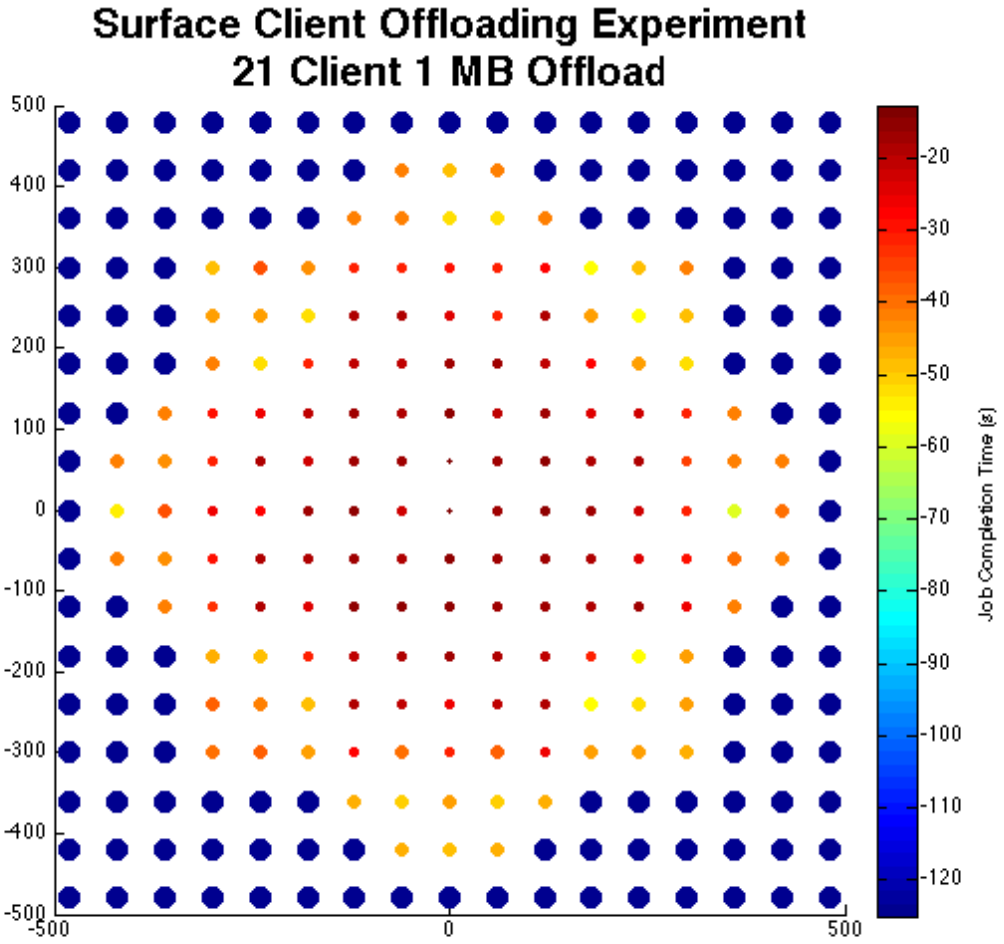

Figure 43. Each circle represents the average client run time for all client jobs when the server is located at that position. Longer running jobs are larger and more blue in color. The large dots on the perimeter are server locations that are out of communication range of the clients. In these locations, computation is performed locally.

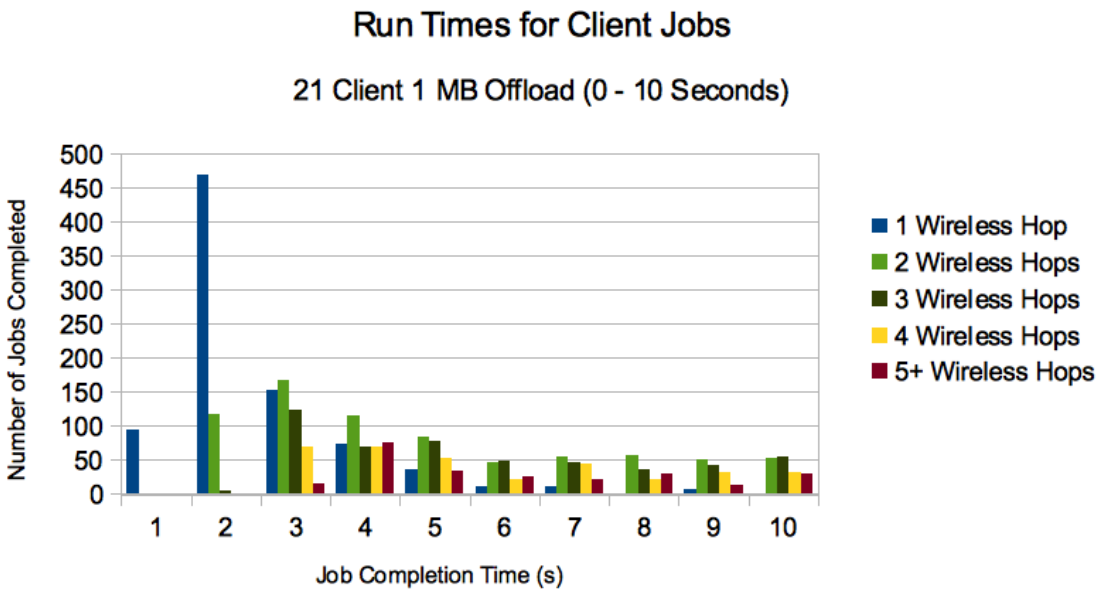

Figure 44. The histogram shows the number of jobs finishing between 0 and 10 s for the 21 client, 1 MB offload experiment.

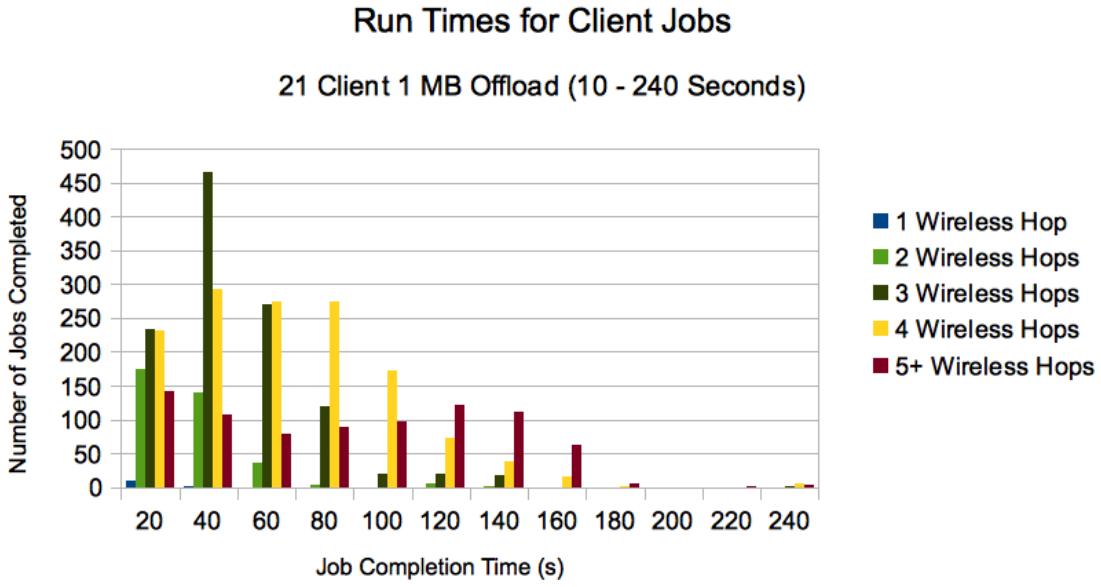

Figure 45. The histogram shows the number of jobs finishing between 10 and 240 s for the 21 client, 1 MB offload experiment. Note the difference in scale on the x axis.

4.2.4 Discussion

The experiments performed demonstrate that there is a clear benefit to offloading. Overall job completion time was reduced when the clients were offloading. However, clients do start to experience longer wait times as the number of clients and the size of the data being offloaded increased. The biggest cause of this delay is due to interference. This is shown by the large disparity in the histograms between 1 KB and 100 MB. Even though there is a large disparity between these sizes, the transmitters are configured to transmit any data it has at 54 Mbps, so that a noncongested network the transmit time for a hop should be well under a second. However, even in cases where the most interference was occurring, figures 44 and 45, a large percentage of the jobs were able to finish faster by offloading than they would have by local computation. Another important thing to consider is that in these experiments, all of the clients start submitting jobs within the first second of simulation. Depending on the scenario, that may or may not be a realistic amount of network congestion.

The results also show the importance of positioning the server. As figure 43 shows, the average run time for all clients is the lowest when the server is centrally located. Specifically, for the 21 client, 1 MB experiment, the maximum job run time for the center position was approximately 45 s compared to approximately 240 s on the edge of the network. Therefore, an important goal when positioning the server should be to minimize the number of wireless hops between the client and server.

As mentioned previously, TCP is a reliable transport protocol in that a data stream transferred from the client to the server is guaranteed to arrive at the server as an exact copy from what left the client. One of the mechanisms that TCP uses for this is a timeout value that will go off if a packet sent is not ack'd within a specific period of time. One feature of TCP is that this timeout value doubles every time a packet is resent. Therefore, when a large number of packet drops occur, like during periods of heavy network congestion, this value can become large and can have a significant effect on client run times.

Following is an excerpt from an ns-3 TCP log for a long running client job. The initial timeout value is 200 ms and doubles each time the timeout expires.

```
1 48.0424 Send packet, schedule timeout at 48.2424
2
3 48.2424 No ack received. Timeout. Congestion window reduce to 536.
4 Resend packet. Schedule time out at 48.6424.
5
6 48.6424 No ack received. Timeout. Congestion window reduce to 536.
```

```
Resend packet. Schedule time out at 49.4424.

49.4424 No ack received. Timeout. Congestion window reduce to 536.
Resend packet. Schedule time out at 51.0424.

51.0424 No ack received. Timeout. Congestion window reduce to 536.
Resend packet. Schedule time out at 54.2424.

54.2424 No ack received. Timeout. Congestion window reduce to 536.
Resend packet. Schedule time out at 60.6424.
```

In this example, the client receives an ack at 55.2341 s and the transfer continues. An important point here is that when a timeout occurs, the amount of data that the client will send before receiving an ack, called the TCP congestion window, is reduced. In the above example the window has been reduced to one segment of 536 bytes. Therefore, between 48.0424 and 55.2341 s, the client has only successfully sent 536 bytes to the server. Now that it has received an ack, the window will begin to grow again. Due to the amount of traffic that is being sent in some of these experiments, delays like this can be common and are the cause of some of the long client job times.

5. Conclusions

In this work we presented an application framework that allows for testing of computation offloading from mobile devices, such as smart phones, to vehicular-mounted HPCs in battlefield scenarios. Use of computation offloading can provide the Warfighter with a new array of tools all accessible from one lightweight device. Another added benefit of computation offloading is that it can be used to extend battery life by offloading computationally expensive, and battery draining, calculates to a remote machine.

Using the ns-3 models we created, we performed a variety of experiments that studied aspects of computation offloading algorithms including computation offloading algorithms, HPC placement, and network topology. Initial results indicate that offloading computationally expensive jobs to mobile HPCs can increase the utility of mobile devices as computation platforms in the battlefield.

6. References

1. Ha, K.; Lewis, G.; Simanta, S.; Satyanarayanan, M. Cloud Offload in Hostile Environments. 2011.

2. Chun, B.-G.; Ihm, S.; Maniatis, P.; Naik, M.; Patti, A. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, ACM: New York, NY, USA, 2011.

3. Kosta, S.; Aucinas, A.; Hui, P.; Mortier, R.; Zhang, X. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, IEEE: Orlando, Florida USA, March 2012.

4. Amazon ec2. http://aws.amazon.com/ec2/, August 25, 2013.

5. Shires, D.; Henz, B.; Park, S.; Clarke, J. Cloudlet Seeding: Spatial Deployment for High Performance Tactical Clouds. In *PDPTA'12*, Worldcomp: Las Vegas, Nevada, USA, July 2012.

6. Henderson, T. R.; Roy, S.; Floyd, S.; Riley, G. F. ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, ACM: New York, NY, USA, 2006.

7. Jacquet, P.; Muhlethaler, P.; Clausen, T.; Laouiti, A.; Qayyum, A.; Viennot, L. Optimized link state routing protocol for ad hoc networks. In *Multi Topic Conference, 2001. IEEE INMIC 2001. Technology for the 21st Century. Proceedings. IEEE International*, IEEE Incorporated, Lahore Section and Lahore University of Management Sciences (LUMS): Lahore, Pakistan, 2001.

8. Sookoor, T.; Doria, D.; Bruno, D.; Shires, D.; Swenson, B.; Pollock, L. Utilization-aware Computation Offloading in Wireless Ad-Hoc Networks. 2013.

9. lpsolve. http://lpsolve.sourceforge.net/5.5/, June 1, 2013.

10. Stoffers, M.; Riley, G. Comparing the ns-3 Propagation Models. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, IEEE: Arlington, VA, 2012.

11. Inc., A. Apple Introduces iPhone 5. http://www.apple.com/pr/library/2012/09/12Apple-Introduces-iPhone-5.html, 2012.

7. List of Symbols, Abbreviations, and Acronyms

ACK	Acknowledgement
AODV	Ad-Hoc On-Demand Distance Vector Routing
CPU	Central Processing Unit
CSMA	Carrier Sense Multiple Access
DSR	Dynamic Source Routing
GPU	Graphics Processing Unit
HPC	High Performance Computer
IP	Internet Protocol
MANET	Mobile Ad-Hoc Network
OFDM	Orthogonal Frequency-Division Multiplexing
OLSR	Optimized Link State Routing
OSI	Open Systems Interconnection
s	seconds
TCP	Transmission Control Protocol
UML	Unified Modeling Language

NO. OF COPIES	ORGANIZATION
1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
6 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIH S D DORIA T SOOKOOR D SHIRES D BRUNO R TAYLOR S PARK

INTENTIONALLY LEFT BLANK.